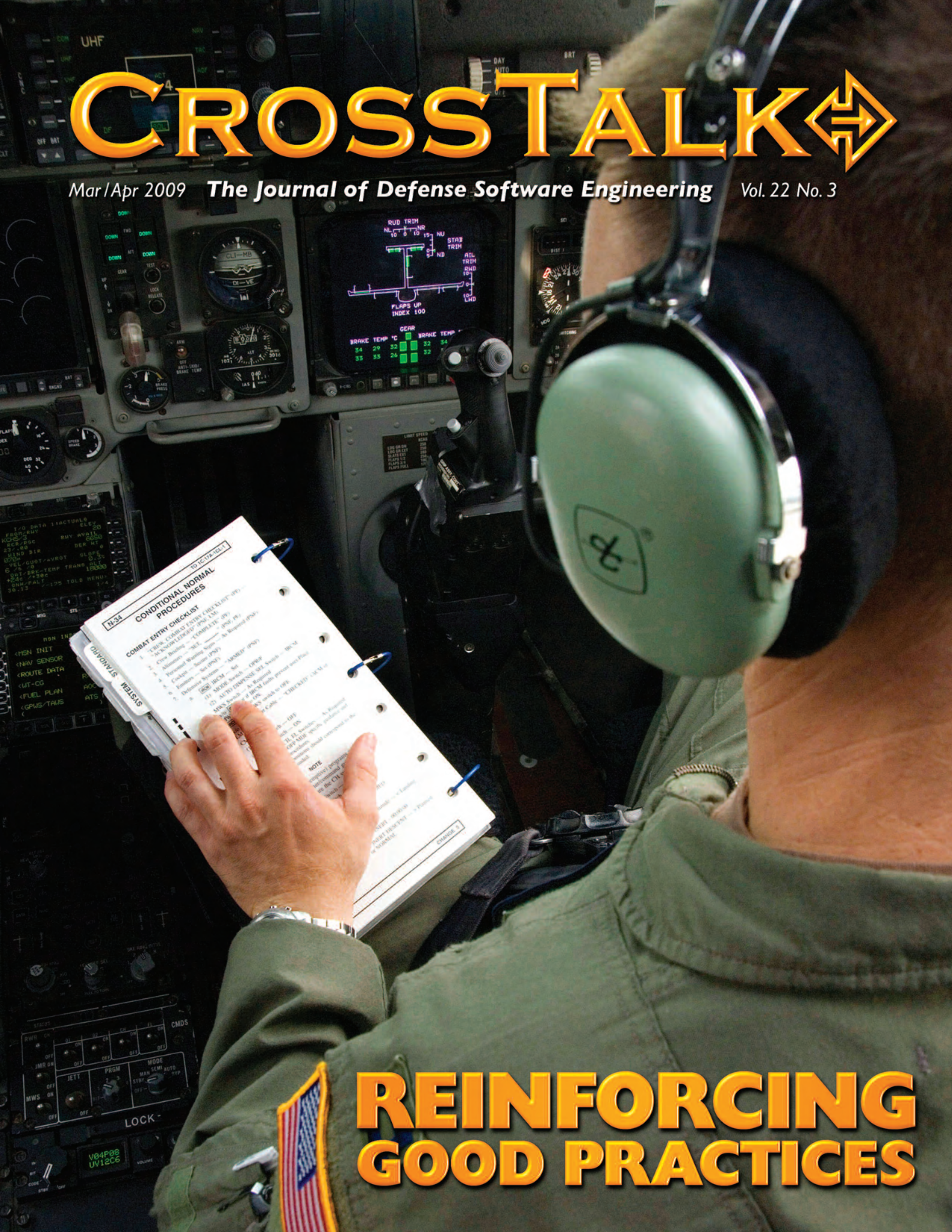
CrossTalk

Mar/Apr 2009 The Journal of Defense Software Engineering Vol. 22 No. 3



Reinforcing Good Practices

Reinforcing Good Practices

4 **Software Assurance Practice at Ford: A Case Study**
The authors examine how software assurance best practices at the Ford Motor Company help develop and maintain the organization's software assets.
by Dr. Nancy R. Mead, Dr. Dan Shoemaker, and Jeffrey A. Ingalsbe

8 **Defect Detection By Developers**
While testers usually detect defects, Rao looks at how developers, with minimal extra effort, can provide easier and more enhanced detection.
by D.T.V. Ramakrishna Rao

12 **A Uniform Approach for System of Systems Architecture Evaluation**
The authors examine a uniform evaluation approach that focuses on SoS quality attribute considerations early in the development life cycle.
by Michael Gagliardi, William G. Wood, John Klein, and John Morley

16 **Static Analyzers in Software Engineering**
This article outlines the strengths and limitations of static analyzers, and explains why they should be a key part of every software development process.
by Dr. Paul E. Black

18 **Management's Inspection Responsibilities and Tools for Success**
This article looks at software inspection pitfalls, upper management's responsibilities during inspection, and computerized inspection tools for each stage of the process.
by Roger Stewart and Lew Priven

Software Engineering Technology

23 **The Evolution of Software Size: A Search for Value**
Minkiewicz looks back at 25 years of software sizing methodologies and suggests what we can learn from the past.
by Arlene F. Minkiewicz

Open Forum

27 **Understanding Software Project Estimates**
Learn more about formal estimation techniques, software estimation tools, the misinterpretation of estimation as target setting, and the accuracy of early estimates.
by Katherine Baxter

Departments

3 **From the Publisher**

11 **Coming Events**

17 **Call for Articles**

22 **Web Sites**

30 **SSTC Ad**

31 **BackTalk**

On The Cover
Cover Design by Kent Bingham

Additional art services provided by Janna Jensen

CrossTalk

OSD (AT&L)	*Kristin Baldwin*
NAVAIR	*Joan Johnson*
309 SMXG	*Karl Rogers*
DHS	*Joe Jarzombek*
Managing Director	*Brent Baxter*
Publisher	*Kasey Thompson*
Managing Editor	*Drew Brown*
Associate Editor	*Chelene Fortier-Lozancich*
Publishing Coordinator	*Nicole Kentta*
Phone	(801) 775-5555
E-mail	stsc.customerservice@hill.af.mil
CrossTalk Online	www.stsc.hill.af.mil/crosstalk

CrossTalk, The Journal of Defense Software Engineering is co-sponsored by the Office of the Secretary of Defense (OSD) Acquisition, Technology and Logistics (AT&L); U.S. Navy (USN); U.S. Air Force (USAF); and the U.S. Department of Homeland Security (DHS). OSD (AT&L) co-sponsor: Software Engineering and System Assurance. USN co-sponsor: Naval Air Systems Command. USAF co-sponsor: Ogden-ALC 309 SMXG. DHS co-sponsor: National Cyber Security Division in the National Protection and Programs Directorate.

The USAF Software Technology Support Center (STSC) is the publisher of CrossTalk, providing both editorial oversight and technical review of the journal. CrossTalk's mission is to encourage the engineering development of software to improve the reliability, sustainability, and responsiveness of our warfighting capability.

S T S C

Subscriptions: Send correspondence concerning subscriptions and changes of address to the following address. You may e-mail us or use the form on p. 29.

517 SMXS/MXDEA
6022 Fir AVE
BLDG 1238
Hill AFB, UT 84056-5820

Article Submissions: We welcome articles of interest to the defense software community. Articles must be approved by the CrossTalk editorial board prior to publication. Please follow the Author Guidelines, available at <www.stsc.hill.af.mil/crosstalk/xtlkguid.pdf>. CrossTalk does not pay for submissions. Published articles remain the property of the authors and may be submitted to other publications. Security agency releases, clearances, and public affairs office approvals are the sole responsibility of the author and their organizations.

Reprints: Permission to reprint or post articles must be requested from the author or the copyright holder and coordinated with CrossTalk.

Trademarks and Endorsements: This Department of Defense (DoD) journal is an authorized publication for members of the DoD. Contents of CrossTalk are not necessarily the official views of, or endorsed by, the U.S. government, the DoD, the co-sponsors, or the STSC. All product names referenced in this issue are trademarks of their companies.

CrossTalk Online Services: See <www.stsc.hill.af.mil/crosstalk>, call (801) 777-0857 or e-mail <stsc.web master@hill.af.mil>.

Back Issues Available: Please phone or e-mail us to see if back issues are available free of charge.

Reinforcing Good Practices

CrossTalk would like to thank the Department of Homeland Security for sponsoring this issue.

We all have been taught sound practices since childhood. Remember the ol' dental mantra of "don't forget to brush your teeth after each meal"? Those instructions were soon augmented with a warning that brushing alone was not enough, and that flossing and regular check-ups were needed to reinforce brushing and prevent the development of dental maladies. Our experience with these routines over our lifetime confirms the worth of reinforcing good, basic practices. Even armed with this knowledge, many of us at some point—then, now, or along the way—ignored the act of reinforcement and suffered the occasional, painful cavity.

Software practices are similar because we understand the value of implementing well-defined best practices, code reviews, and well-structured architectural design in combination with the basics. Even with that understanding, the drive for on-time delivery or budget and time constraints hampers the opportunity to perform the reinforcing actions that prevent future problems—and we suffer the software equivalent consequence, otherwise known as software defects.

The March/April issue of CrossTalk provides five well-crafted articles intended to assist developers in avoiding the pain of "software cavities" by bolstering their current processes through implementation of sound reinforcement practices. Dr. Nancy R. Mead, Dr. Dan Shoemaker, and Jeffrey A. Ingalsbe share best software assurance practices as developed by the Ford Motor Company in *Software Assurance Practice at Ford: A Case Study*. D.T.V. Ramakrishna Rao increases developer's defect awareness through a reinforcing practice known as "active reading" in *Defect Detection by Developers*. SEI authors Michael Gagliardi, William G. Wood, John Klein, and John Morley offer a consistent approach for evaluating and mitigating risk and challenges to large-scale systems in *A Uniform Approach for System of Systems Architecture Evaluation*. In *Static Analyzers in Software Engineering*, Dr. Paul E. Black contrasts the strengths of static analyzers with testing as a method for detecting possible code problems. Roger Stewart and Lew Priven explore ways for leaders to make software inspections unassailable in *Management's Inspection Responsibilities and Tools for Success*.

CrossTalk offers two additional articles this month that complement our theme quite well. In *The Evolution of Software Size: A Search for Value*, Arlene F. Minkiewicz, through her own industry experiences, analyses 25 years of efforts to solve the problems of software size. As well, Katherine Baxter offers up her article, *Understanding Software Project Estimates*, in an effort to remind us of the virtues of software cost estimating on each and every project undertaken.

In closing, remember the words of English philosopher and scientist Francis Bacon who once said that it is "... not what we profess but what we practice that gives us integrity." While Sir Francis wasn't referring to software when he made this statement, software integrity and quality can only increase when we follow his advice and reinforce our already good practices.

Kasey Thompson

Kasey Thompson
Publisher

Notice a Few Changes?

Beginning this month and for the remainder of 2009, CrossTalk will be published every other month. CrossTalk issues in May/June, July/August, September/October, and November/December will be larger in volume with additional articles. Our hope is that this format will allow for more detailed articles in a format more conducive for defense software engineering. Also, we will be adding some new elements, beginning with the sponsor logo box at the top of this page. Look for more changes as we begin to incorporate them throughout the next few issues.

Software Assurance Practice at Ford: A Case Study

Dr. Nancy R. Mead
Software Engineering Institute

Dr. Dan Shoemaker
University of Detroit Mercy

Jeffrey A. Ingalsbe
Ford Motor Company

Software pervades our technological society, handling our financial transactions, managing power transmission, facilitating most forms of communication, and keeping us safe. This makes defects in software one of the most potent threats to our national security, and turns identification of best practices in software development, acquisition, and long-term use the highest national priority. This article presents the best practices employed by the Ford Motor Company to develop and maintain their software assets.

Defects in software are among the most potent threats to our national security. That is because these defects—whether by malicious agent placement or faulty manufacturing—represent avenues of attack for any potential criminal, terrorist, or enemy adversary.

Therefore, it is the highest national priority to find and employ the right set of practices in software development, acquisition, and long-term use that will prevent those defects. This is a noble goal, but the fact is that there is very little agreement on what exactly is the optimum set of best practices. And worse, we have almost no idea what the current state-of-the-practice is in business and industry.

Essentially, there are no concrete points of reference to guide us in affecting change to the noticeably unsuccessful way that industry has currently approached the problem. It's like "Alice in Wonderland" when Alice doesn't care where she goes. The Cheshire Cat responds that in getting "somewhere" that "it doesn't matter which way you go ... you're sure to do that ... if you only walk long enough." So it seems like the alternatives are to develop a clear understanding of the current state-of-the-industry best practice, or to continue to do a lot of pointless walking.

One Company's Best Practice Example

With no clear practices currently in place, a case study of current industry practice, especially from a *Fortune* 10 company, is an extremely valuable and useful tool for people who are interested in changing the state of practices in the software community. This article presents an overview of the control processes employed by the Ford Motor Company to develop and maintain their software assets, a brief history of those processes to provide context, and a discussion of their future.

What Has Gone Before

The IT Security and Controls organization at Ford dates back to 1998, years before an agreed-upon international standard for information security management systems like ISO/IEC 27001:2005 [1] existed. Since that time, the organization has developed control processes for applications (first) and infrastructure (second). These are the application control review (ACR) process and the infrastructure control review (ICR) process, respectively. Later, they added a control process called systems control review (SCR), to be conducted yearly to assess the effectiveness of controls that had been specified by the ACR or ICR process and to address the results of internal audits. Ethical hacking and static code analysis (on select applications or infrastructure) were instituted to identify vulnerabilities (read defects) in code that was already written. Threat modeling (on select applications and infrastructure) was instituted to prevent vulnerabilities before code is written. Finally, the data from ethical hacking and static code analysis was fed back into a training and awareness program to help developers understand common errors being inserted into code.

Current State Control Processes

Ford's current control processes span the entire asset life cycle (software or hardware) from conception to retirement. They are the ACR process, the ICR process, threat modeling, ethical hacking, static code analysis, risk assessment, and the SCR process. As a set, these processes work together to ensure the overall security and integrity of Ford's software. Figure 1 shows when these processes are typically executed in the life cycle but does not show how often they are executed. For example, all applications must compose application control documentation using the ACR process before launch, but not all applications are threat modeled. Threat modeling is performed only on those applications that score highest on a risk assessment or are deemed strategically important (a small subset of the entire portfolio). Ethical hacking is performed based on an independent assessment of the criticality of the system or infrastructure. After launch, a yearly risk assessment is performed that determines whether a more detailed SCR is required. All applications and infrastructure scoring high and a semi-random sample of those scoring medium or low are required to perform the SCR. The following sections give an overview of the processes used.

ICR Process

The ICR process is responsible for ensuring that the overall IT infrastructure is correct and that adequate controls exist.

Any system activity planning to use a piece of infrastructure must ensure that an ICR has been done prior to its inclusion in the application design. More importantly, a reference to the infrastructure's ICR must be included in the ACR of every application using the infrastructure.

The ICR is initiated by an asset owner and performed by a designated internal control coordinator (ICC), with assistance from a security control champion (SCC). Next, a meeting is conducted with the person who is accountable for managing any identified risk (the infrastructure owner). The meeting might also include the technical support staff or subject matter experts of the infrastructure owner. For purchased or commercial off-the-shelf solutions, the vendor's technical support staff may be included as part of the infrastructure team. In this meeting, infrastructure components and the risks applicable to them are identified and a *risk matrix* is developed.

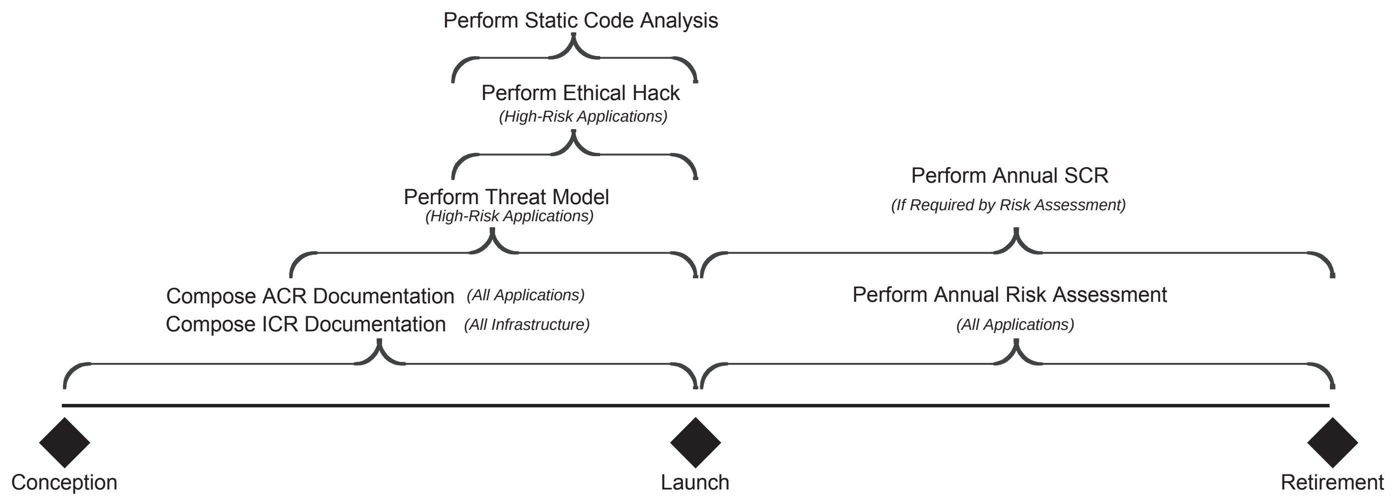

Perform Static Code Analysis
Perform Ethical Hack
(High-Risk Applications)
Perform Threat Model
(High-Risk Applications)
Perform Annual SCR
(If Required by Risk Assessment)
Compose ACR Documentation (All Applications)
Compose ICR Documentation (All Infrastructure)
Perform Annual Risk Assessment
(All Applications)
Conception
Launch
Retirement

Figure 1: *Ford's Control Processes*

The risk matrix is then used to develop the ICR package. The infrastructure owner, ICC, and SCC prepare this package. It includes an overview of the technology being employed, a network diagram, and a data/process flow diagram. The two diagrams document where the infrastructure is being deployed, identify the hardware used to deploy it, and show how data flows through the infrastructure. The infrastructure risk matrix is also cross-referenced to the tangible controls that have been put in place to address each threat.

Finally, all infrastructure components are explicitly itemized along with all threats included in the infrastructure risk matrix. At this point, the controls used to reduce the risk of a specific threat to that component are itemized in terms of who is performing the action, what they are doing, and what threat is being reduced by performing that action. A specific disaster recovery plan is also specified along with the roles and responsibilities for the critical job functions needed to implement and support the infrastructure once a change has been made.

ACR Process

The ACR process operates at the level of individual applications, which is the level that would be of the most interest to people concerned with secure software assurance best practices.

The basic goal of the ACR process is to reduce the risk associated with information technology applications. It does that by ensuring that appropriate controls are implemented for each application and that those controls are functioning properly.

The ACR process assesses all significant new and changing services, processes, operations, and control processes. The ACR process applies to all IT applications, including commercial off-the-shelf, independent of whether the application resides internally (on the internal Ford network) or externally (hosted by an external provider).

ACR/ICR Roles and Responsibilities

The ACR and ICR processes involve the application/infrastructure owner, the IT organization, a designated ICC, an SCC, and (sometimes) an auditor. Telecommunication services is involved for external facing assets and externally hosted assets.

The application or infrastructure owner is responsible for ensuring that adequate controls exist and that the controls mitigate risk at a reasonable cost. The IT organization is responsible for assisting the application or infrastructure owner in defining adequate controls and incorporating them into the application or infrastructure. The ICC is responsible for overseeing the ACR and ICR for the relevant application data, programs, and infrastructure. The ICC is responsible for reviewing and approving the application classification and also provides advice on business controls and coordinates the control review. The SCC is responsible for guiding application owners in performing the specified ACR and ICR processes and also assists with the completion of the ACR/ICR documents (as required). The SCC conducts risk assessments and pre-implementation reviews for compliance with corporate policies and verifies that the documentation is valid based on current technical and policy information.

ACR Process Steps

The first step in the ACR process is to classify the application using the three basic security objectives: confidentiality, integrity, and availability (CIA). The application is classified along a sliding scale from low to high impact (numerically from 1 to 3) on each of these objectives[1]. From the CIA rating, a list of questions is automatically generated. Each question addresses a particular control.

The application owner, IT organization, SCC, and ICC work together in a series of meetings to prepare the materials required for completion of the ACR. The required materials include a non-technical description of the application that describes the purpose of the application, how it will function when installed, products, hardware, and software needed, and what activities use the application. The review materials also include a process flowchart of the application that identifies and describes the sources of input, master files, outputs, and major processing programs. It must describe the application to someone not familiar with it and graphically highlight where controls are required. Also required is a network flowchart, which is a detailed depiction of the network and the various connected host computers. The aim of this flowchart is to facilitate an understanding of the controls architecture. It depicts the flow of application data and the direction of the data flow. If connectivity is required from an outside vendor, the diagram should also depict the access-control points (firewalls, routers, or virtual private network devices) and the network flow though these devices.

Once the materials are complete, the ICC will determine that the ACR is ready for a formal review. The product of the formal review is a statement concerning the adequacy of controls. The application owner is then responsible for revising the controls and corresponding ACR documentation based on the outcome of this review.

SCR Process

The SCR process is performed annually on those applications or infrastructure that score *High* on the annual risk assessment as well as a semi-random sample of medium- and low-risk applications or infrastructure. It comprises a set of checklists, testing, and evidence-gathering techniques that (together) are used to assess whether all of the controls that were documented by the ACR and ICR processes are being performed. It essentially validates that the planned controls are working and that additional controls are put in place if anything has changed since the last review.

The IT Policy Manual, which guides all Ford IT work, requires an annual risk assessment for each application and infrastructure component. Ford requires this in order to prioritize SCR process work each year. The SCR process ensures that all of the necessary controls are adequate and are functioning effectively. This helps managers to identify control weaknesses before they can be exploited. It also provides a mechanism and process for developing corrective actions and tracking closure of the weaknesses that may be found.

Threat Modeling

Threat modeling is performed on strategically important projects and those that score high on a risk assessment. It is typically performed in the design phase and is focused on prevention of vulnerabilities that may have a high business impact if exploited. In its simplest form, threat modeling involves six steps:

1. Identifying assets within the application or infrastructure (e.g., data or processes).
2. Identifying people involved (e.g., administrators or users).
3. Identifying high-level or meta-use cases (these are not the same as Unified Modeling Language use cases).
4. Identifying threats to assets.
5. Classifying threats using what is called DREAD—**D**amage potential, **R**eproducibility, **E**xploitability, **A**ffected users, and **D**iscoverability [2]—and then assigning a 0 to 5 value to each of the five types[2].
6. Choosing whether to accept, transfer, avoid, or mitigate the risk posed by those threats.

Threat modeling typically takes four to five working sessions, and should involve both IT (because they understand threats) and the (internal) business customer (because they understand value).

Ethical Hacking

Ethical hacking is performed on strategically important projects and by request. It is typically performed after implementation and is focused on the detection of vulnerabilities. The ethical hack team uses a combination of tools, processes, and acquired skills to perform the hacks. Detailed metrics are kept on the results of the hack and given to the requestor. Metrics on classes of vulnerabilities (e.g., buffer overflow, cross-site scripting, Structured Query Language injection) have been used to develop a training course for developers.

"With no clear practices currently in place, a case study of current industry practice ... is an extremely valuable and useful tool for people who are interested in changing the state of practices in the software community."

Static Code Analysis

Static code analysis is performed by request when an application owner feels that there is a higher risk associated with the project. It is typically performed after implementation and is focused on the detection of vulnerabilities. A commercial off-the-shelf tool is used to analyze the code. False positives are eliminated and the report is given back to the requestor.

Future State

The control processes outlined in this article have been instituted over a period of 10 years. The older processes have an information assurance (IA) *feel* to them (i.e., information protection) while the newer processes have a software assurance feel (i.e., prevention of coding vulnerabilities). The acceptance and popularity of ISO/IEC 27001 and the desire to make decisions based on business value and risk has prompted Ford's IT Security and Controls organization to begin aligning their processes with the international standard. The work is in its infancy, but it is already clear that quantifying asset value and risk—and using them to make decisions—is the right course.

Observations

This is a case study, so conclusions should not be drawn. However, some important observations can be made based on what has been reported here.

First, Ford clearly pays considerable attention to the security of its applications and its associated infrastructure. The degree of detailed rigor of their processes, the time spent, the documentation produced, and the obvious bureaucratic coordination and control requirements support that observation. Additionally, they pay particular attention to anything that might fall under Sarbanes-Oxley Act[3] purview by requiring that those applications are always addressed with the highest degree of rigor. Their control perspective, until recently, was oriented toward classic IA principles rather than those of software assurance. Specifically, the risks are rated on the CIA scale, which is typical of IA risk assessments. While IA concerns are to some extent founded on secure software assurance, the things that threaten information are not the same as those things that might threaten software. In fact, it is perfectly possible for a piece of software, both under development and in actual use, to have all of the necessary controls to assure any set of regulatory requirements while still being full of exploitable holes.

Second, anecdotal evidence suggests that there may be an evolutionary progression an organization goes through as they move from *no security focus* to *best in class*. First, they get things like antivirus, incident response, and forensics in place. These things are served up centrally and (of course) have an IA focus. Next, they move on to securing applications and infrastructure. However, since they have probably not plugged into those organizations yet, it is natural for them to show up just before launch or at gate review to render their opinion on the security of the application or infrastructure. This would tend to be adversarial. Next, they would get involved early in the software development process and help developers classify the importance of the information being created/deleted/modified/transmitted by the application. They would then prescribe controls based on that score and do things like ethical hacking to ensure that those with the highest scores don't have big vulnerabilities. Some companies stop at this point. More introspective companies may begin to engage their development community in order to build better software and ask questions like, "Is there a need to protect things at different levels?" Such companies would then do things like look at their penetration test results and take the top 10 vulnerabilities back to the development community, then working with them to change coding practices in order to eliminate those vulnerabilities and establish better coding practices. This is where Ford is right now. They have rolled out threat modeling and are engaging their development community in order to develop more secure coding practices.

The fact that a company well-known for its competence and effectiveness in IT security has recognized (and is acting upon) the need to move beyond IA to software assurance bodes well for others who are not nearly as far along the evolutionary scale. However, it may indicate that we have a ways to go in popularizing the importance of software assurance best practice in conventional IT security operation.

It is possible and (perhaps) probable that many of the practices that the software assurance community would recognize as secure software assurance are taking place on an ad-hoc basis within the actual application development and maintenance function within IT itself, which suggests where the next study should be focused. However, it is also clear that the only way to ensure that those practices are institutionalized is to provide both the incentive and the guidance to get large companies to implement secure software assurance practice as part of their conventional application security operation.◆

References

1. ISO/IEC 27001:2005. "Information Technology – Security Techniques – Information Security Management Systems – Requirements." Distributed through the American National Standards Institute. 23 Aug. 2007.
2. Howard, Michael, and David LeBlanc. Writing Secure Code (With CD-ROM). Microsoft Press, 2001.

Note

1. For example, an application that handles the most confidential data with the highest integrity requirement and the highest availability requirement would be rated 3,3,3. An application that handles the least confidential data with the lowest integrity requirement and the lowest availability requirement would be rated 1,1,1.
2. By assigning a 0 to 5 value for each part of DREAD, a final risk value is obtained that allows threats to be compared. Despite the fact that a *formula* is used to calculate a risk value, it is important to understand that it is still subjective. A significant effort by participants needs to be made to be consistent.
3. The Sarbanes-Oxley Act of 2002 (Pub.L. 107-204, 116 Stat. 745, enacted 30 July, 2002) is a federal law that describes specific mandates and requirements for financial reporting.

About the Authors

Nancy R. Mead, Ph.D., is a senior member of the technical staff in the Networked Systems Survivability Program at the SEI. She is also a faculty member at Carnegie Mellon University. Mead's research interests are in the areas of information security, software requirements engineering, and software architectures. She is a Fellow of the IEEE and the IEEE Computer Society and is also a member of the Association for Computing Machinery. Mead received her doctorate in mathematics from the Polytechnic Institute of New York, and received bachelor's and master's degrees in mathematics from New York University.

SEI
4500 5th AVE
Pittsburgh, PA 15213
E-mail: nrm@sei.cmu.edu

Dan Shoemaker, Ph.D., is the director of the Centre for Assurance Studies. He has been professor and chair of computer and information systems at the University of Detroit Mercy for 24 years, and co-authored the textbook, "Information Assurance for the Enterprise." His research interests are in the areas of secure software assurance, information assurance and enterprise security architectures, and information technology governance and control. Shoemaker has both a bachelor's and a doctorate degree from the University of Michigan, and master's degrees from Eastern Michigan University.

Computer and Information Systems – College of Business Administration
University of Detroit Mercy
Detroit, MI 48221
Phone: (313) 993-1202
E-mail: shoemadp@udmercy.edu

Jeffrey A. Ingalsbe is a senior security and controls engineer with the Ford Motor Company. He is involved in information security solutions for the enterprise, threat modeling efforts, and strategic security research. Ingalsbe also serves as an expert industry panelist on two national working groups within the DHS's Cybersecurity Division. He has a bachelor's degree in electrical engineering and a master's degree in computer information systems from Michigan Technological University and the University of Detroit Mercy, respectively. Ingalsbe is currently working on his doctorate degree in software engineering at Oakland University.

Ford Motor Company
17475 Federal DR
STE 800-D04
Allen Park, MI 48101
Phone: (313) 390-9278
E-mail: jingalsb@ford.com

Defect Detection By Developers

D.T.V. Ramakrishna Rao
Infosys Technologies Limited

Poor quality caused by defects continues to be a major problem facing the software industry. Unlike the traditional way of handling this problem where testers detect defects, this article suggests approaches where developers detect defects. This approach builds upon existing techniques, augments certain activities that developers often already engage in, and focuses on detecting defects in existing code. The end result is little additional effort in defect detection, easier fixes, and enhancing the effectiveness of the original intention of the activities.

Bugs plague almost all software systems. Indeed, more than half the time spent in a typical software project is on bug fixing [1]. Given the severe consequences bugs may have and the significant percentage of project effort associated with them, tackling bugs is of fundamental importance.

Defect detection is the primary strategy used to tackle defects, with testing being the predominant way defect detection is accomplished. In the past decade, inspections have been increasingly used to supplement testing. Still, defects present a significant problem. The software industry must continue to find new *cost-effective* ways to supplement current strategies to find defects.

This article proposes new approaches to detect defects. Developers engage in certain activities, often to detect defects in their *new* code. The new approaches are additions to these activities towards detecting defects in the *existing* code. These additions require little extra effort. Since defects in the existing code are detected almost as a by-product of these activities, they are referred to as *by-product defects* (BDs)[1]. This strategy has resulted not only in efficient detection of BDs but also in easier fixing.

I, along with a team at Infosys Technologies, have been applying these approaches for the past eight years on large and complex software systems and have found hundreds of defects.

Some developers use variations of the activities discussed in this article, but they are by no means universally used. Also, we have not seen a clear articulation of them in research literature related to software development. Hence, description of these activities can be construed as a description of a set of best practices for software developers. The primary contribution, however, is the addition made to these activities towards detecting BDs.

The primary sections of this article:

- Identify defect consciousness, an important ingredient for detection of BDs.
- Describe the activities during which developers find BDs.
- Relate experiences in deploying the activities.

Defect Consciousness

By defect consciousness, we mean the understanding of defects: what they are, what their types are, how they show up, what causes them, etc. Novices tend to lack this knowledge as programming courses generally do not emphasize this knowledge vis-à-vis writing programs. Developers normally gain this knowledge with experience, but not consciously. That leads to gaps in their knowledge. Hence, it is important to gain defect understanding consciously.

I found certain approaches to be rather effective in cultivating defect consciousness. Some books and papers focus on defects (for example, [2]), and are useful for a general understanding of defects. This understanding should be supplemented with defects in specific projects that developers are working on, as there are often defect types idiosyncratic to a project. Projects often have coding and other guidelines in this regard. While guidelines tend to be just bland statements, I found that augmenting a guideline by pointing at a specific fixed bug report (as an illustration) has many benefits. As the saying goes, "an example is better than a precept" [3]. It illustrates a defect in action: how it looks in code, how it shows up during execution, how it is debugged, etc. It helps in recognizing defects easily. Developers should also develop a life-long habit of understanding bug reports fixed by others.

The following activities may detect defects even without defect consciousness, but they will be more effective coupled with it.

Activities to Detect BDs

This section describes a set of activities during which BDs may be detected by developers: reading modules, regular reviews, triggered reviews, regular unit testing, and triggered unit testing. The activities are deployed in a typical industrial software development manner with dozens of developers working on a large and complex software system.

Description of each activity is organized as follows:

- What is the activity?
- When is it done?
- Why is it done?
- How is it done?
- How might BDs be detected as part of it?
- How does detecting BDs help the primary purpose of the activity?
- Why is it easier to fix BDs compared to defects detected by other means (such as traditional testing)?

Reading Modules

A module is a single file or a collection of files used to achieve a specific functionality. Developers should read a module for two reasons:

- **Reading modules for learning.** Reading code is the best (and sometimes the only) way of understanding the functionality implemented by a module. For example, in the case of networking protocols, protocol understanding from standards can be clarified and crystallized by reading the code implementing the protocol.
- **Reading modules while working on a bug or an enhancement.** Developers should completely read those modules that they are modifying as part of their work. It helps in not introducing bugs by making sure that *all* the required changes are made for their work in that module.

Many techniques exist for reading code [4]. A technique particularly effective to detecting BDs is active reading [5], which is a form of critical reading of the code: read a few lines, try to paraphrase them in your own words, ask questions, then try to answer those questions. I have seen in practice that reading a module once is not enough; reading twice is often sufficient. Active reading, coupled with defect consciousness, is very effective in both understanding code (primary purpose) and in detecting BDs.

When a defect is found while reading a module, it is much easier to fix compared to a traditional testing-found defect, for the same reason that inspection-found defects are easier to fix compared to testing-found defects (one can find a defect's location, what kind it is, as well as its cause) [6].

Comparison With Walkthroughs

Code reading (reading a module) as advocated in this article is a form of review, walkthrough, or inspection [6]. But there are notable differences. This section explores these differences and their implications.

Predominantly, code walkthroughs are used as a mechanism to review the implementation done by a developer. So, I will first compare walkthroughs and code reading in the context of a developer working on an enhancement to the code base.

A walkthrough is done to detect defects by a set of reviewers after a developer completes implementation. In contrast, code reading is done to detect defects by the developer during implementation. Hence, defects will be found sooner in code reading.

The intention of the walkthrough is to detect defects in the enhancement, so the developer walks the reviewers through his or her changes of the modules. The walkthrough focuses on the changes to a module per se, whereas in code reading, the focus is on the entire module even when only a few changes are made to it. The change in focus helps code reading by detecting even more defects than walkthroughs would. A module in a system might have gone through many modifications over time. As a module evolves, it tends to lose unity, becomes more complex, and hinders maintainability. Because of such evolution, some bugs tend to get introduced. When reading the whole module, there is a higher possibility of detecting those bugs.

Less often, code walkthroughs cover entire modules for special objectives (e.g., security audits). Code reading mainly differs in the way it is structured to efficiently and simultaneously achieve a novel combination of objectives: critical understanding of a module, detection of defects in the module (aided by critical understanding and defect consciousness), and ensuring that changes the developer is making to the module are complete and consistent (again, aided by critical understanding and defect consciousness). I am unaware of an existing code walkthrough that achieves the same objectives.

Regular Reviews

Reviews or inspections are used in some organizations to find bugs in the submitted artifacts [6]. During code reviews, reviewers try to find bugs in the submitted code changes with the help of additional documentation such as checklists and source documents (e.g., designs and requirements of the code changes). But during reviews, they may find bugs not only in the code changes but also in other documents. For example, Gilb [6] observed that inspections often find bugs in the source documents.

"A walkthrough is done to detect defects by a set of reviewers after a developer completes implementation. In contrast, code reading is done to detect defects by the developer during implementation."

I am not, however, aware of any prior observations that reviews may find bugs in code that is not part of the changes. But indeed, reviews are helpful in finding such bugs. If code changes show that a function is modified, reviewers should read related code surrounding the changes, functions that call the changed function, and functions called by the changed function. Reviewers need to understand the related code and, in light of that understanding, check whether the code changes have any bugs. For a reviewer with defect consciousness, the process of understanding the related code using active reading (as previously discussed) provides opportunities for uncovering bugs in that code and help in a more effective review.

BDs found during regular reviews, just like BDs found during reading modules, will be easier to fix.

Triggered Reviews

The reviews in the previous section are conducted before checking code changes into the code base; in some instances, however, there is a need to review code changes afterwards. Suppose you are making changes to your private copy of a code base while working on a bug or an enhancement. When multiple developers are working on the same code base, what you are doing may be affected by what others are checking into, in turn necessitating further changes. Therefore, you should go through each of the check-ins and review those that are related to your changes carefully.

Unfortunately, it is not always easy to know when a check-in is *related* to your change. From experience, I've found the following check-ins require careful review:

- **Check-ins that modify the modules you've changed.** It is very important to scrutinize such check-ins, as they are very likely to affect your changes. You are also in a good position to review those check-ins because of the familiarity with the changed modules (having followed the first activity: reading modules).
- **Check-ins that modify the subsystems you've changed.** Large complex systems are normally divided into subsystems, which in turn are divided into modules. For example, in a networking system, transmission control protocol implementation may constitute a subsystem. If a check-in modifies the subsystem you are changing, it is likely to affect your changes.
- **Check-ins that modify the subsystems you depend upon.** Software systems normally have a set of subsystems that are utilitarian in nature. They are often organized as libraries (e.g., a string-processing library). The rest of the system uses these subsystems. Changes to these subsystems tend to be rare but are not totally unheard of. If a check-in modifies such a subsystem that you are using, you need to update your changes. Moreover, all the changes to utility subsystems should be studied for continuing education on the project, as these subsystems are frequently used.
- **Check-ins related to your subproject.** Large enhancements tend to be implemented by multiple developers as subprojects. If your changes are part of a subproject, you should actively review all of the check-ins in the sub-

Activity	No. of Bugs	Percent
Reading Modules	32	48
Regular Reviews	7	10
Triggered Reviews	12	18
Regular Unit Testing	10	15
Triggered Unit Testing	6	9
Total	**67**	**100**

Table 1: *Bugs Found by the Activities*

project for three reasons. First, the check-in may be directly or indirectly related to your changes. Second, being part of the subproject means that you are in a good position to review the check-ins. Third, most subprojects are such that you may be working on yet another part of the subproject immediately after completing your current part. Hence, it is important to keep track of the design and implementation of the subproject on a continuous review basis.

The first step of reviewing a needed check-in is to understand the code changes made by it, and then assessing its impact on your changes. For a reviewer with defect consciousness, the process of understanding the checked-in code using active reading also provides opportunities for uncovering defects in that code.

BDs found during triggered reviews, just like BDs found during reading modules, will be easier to fix.

Regular Unit Testing

While modifying software, developers can also conduct unit tests to detect bugs in their changes. Their testing can be characterized by two aspects:

- **Tunnel vision.** They tend to concentrate only on the behavior of their changes.
- **Focus on end results.** The tests are often conducted simply to verify the outputs.

This type of unit testing is not very effective in uncovering bugs for two reasons.

First, it is not only important to check the output but also to check the entire processing that led to the output. Sometimes, intermediate processing may be incorrect, but the final result may turn out to be correct. For example, in a program if either function A or B returns true, further processing is undertaken. For a particular input, both the functions should return true but, due to a bug, function B returns false—yet the final result is as expected. Hence, the bug went undetected. These bugs in the intermediate processing may manifest in the future.

The second issue involves changes in large and complex systems: Developers may not be aware of the repercussions of their changes in other parts of the system.

More effective unit testing would take these two points into account. Developers should go through the processing of changed modules in minute detail either using a debugger in single-stepping mode or enabling *tracing* on the modules (a module supports tracing by printing debug output of its processing in great detail). To observe the impact of the changes on the rest of the system, enable the log messages at all levels and assertion checking on the entire system.

Developers should analyze the previously mentioned processing details and the messages produced by the system for anomalies (defect consciousness will aid here). Every anomaly needs to be analyzed to check whether it represents a bug, and (if so) whether it is pre-existing or if it was introduced by the developer's changes. If the anomaly is not a bug, it should enhance the understanding of the system for the developer. If the anomaly is a bug introduced by the developer, it obviously needs to be fixed. If the anomaly is an existing bug, the developer has found the bug and should open a bug report. The report should include the analysis already done to fix the bug faster in the future.

The enhanced unit testing helps both in detecting existing bugs in the system and in more effectively detecting bugs made by a developer.

Triggered Unit Testing

The activity described in this section applies in the same context as described in the Triggered Reviews section: The new check-ins may affect the changes being made. That section suggested reviewing the check-ins in this context. However, reviews may not catch all of the interactions that may exist between the check-ins and your changes. This is when testing becomes useful. Empirically, testing and reviews are shown to be complementary in their defect detection abilities [7].

After merging the check-ins with your changes, do not make any further changes to the code. Run a representative set of passed tests that you previously used for testing your changes (an application of regression testing in a development context). If the test fails, investigate to distinguish between two possibilities: Your code may need to be updated (in light of the check-ins), or there is a bug in the check-ins.

Of course, if your code needs to be updated, do so immediately. And, even if it represents a bug in the check-ins, the investigation already conducted is a good starting point for fixing the bug. Therefore, the complete investigation details should be a part of your bug report in order to fix the bug faster in the future.

Case Study

The described activities have been applied in multiple projects in our organization and have helped in finding hundreds of BDs. To show the benefits of these activities, I present a case study.

This case study shows the results of applying the activities by a single developer in a span of two years while enhancing and fixing bugs on a very large software system. The developer had about five years of experience at the beginning of the case study. The system was a C/C++ based mature networking software system having more than 50 million lines of code and was maintained by more than 100 people. The development process is typical of industrial software development. When a developer is enhancing or fixing a bug, he or she will do unit testing and submit the implementation to peer review. After the review, code is checked-in. When all enhancements and bug fixes for a release are in place, the testing team conducts integration testing and system testing.

Table 1 shows the distribution of BDs detected as a result of applying the activities by the developer. When the developer applied the activities, the detected defects were found in both the developer's new code and the existing code (BDs). Table 1 shows only BDs and not the defects detected by the developer in the new code. The BDs represent defects detected that leaked from the described formal stages of defect detection. For example, BDs detected by triggered review and triggered testing are missed by unit testing and peer review.

In all, the developer detected 67 defects. Reviewing code is found to be particularly effective. Almost half of the defects were detected during reading modules and about 75 percent were detected in some form of reviews. Triggered activities detected 27 percent of the BDs. Two main reasons accounted for their success: division of large enhancements into pieces to be done by multiple developers, and frequent modification of certain modules. Testing activities, however, should not be discounted; testing-detected defects, though fewer, tended to be of higher severity.

The case study shows that, using the activities, a developer can actually detect existing bugs in the system—just like a tester. More importantly, these defects are missed by formal stages of defect detection. The number of defects detected by the developer is of the same order as detected by a test engineer in the same timeframe. It is as if a test engineer was acquired for free!

Conclusion

This article discussed a set of activities for developers to detect defects in their new code, and augmented the activities to detect defects in existing code (BDs). This strategy has resulted in the following advantages:

- Detecting defects with little additional effort.
- Easier fixing of these defects compared to defects found during traditional testing.
- Enhancing the primary purpose of the activities that are augmented to detect BDs.

The current deployment of these activities falls far short of their potential utility. For more effective deployment, this article provides a starting point: Developers should enhance their defect consciousness and follow the activities as described. For long-term retention of these activities, they should be integrated with the development methodologies.

From a larger perspective, this article makes a small contribution regarding how developers may contribute more towards the quality of products. The current development methodologies do not fully utilize the expertise of developers in detecting defects. Proposed here are some strategies to utilize their knowledge. The techniques discussed basically fall into two categories: review and testing. There are many chances/reasons for developers to read or test code. Every such chance should be exploited to detect defects in the existing code, just as was done in this article.◆

Acknowledgment

The author sincerely thanks the CrossTalk Editorial Board, Kasey Thompson, Piyush Jain, Saravana Prasad, Thomas George, and Lilly Vasanthini for their comments on earlier versions of this article.

References

1. McConnell, Steve. Professional Software Development. Boston: Addison-Wesley, 2003.
2. Ploski, Jan, et al. "Research Issues in Software Fault Categorization." ACM SIGSOFT Software Engineering Notes 32(6): 6, 2007.
3. Answers.com. "Example Is Better Than Precept." 2008 <www.answers.com/topic/example-is-better-than-precept>.
4. Laitenberger, Oliver, and Jean-Marc DeBaud. "An Encompassing Life Cycle Centric Survey of Software Inspection." The Journal of Systems and Software 50(1): 5-31, 2000.
5. Clayton, Richard, Spenser Rugaber, and Linda Wills. On the Knowledge Required to Understand a Program. Proc. of the 5th Working Conference on Reverse Engineering. Honolulu, 12-14 Oct. 1998: 69-78.
6. Gilb, Tom, Dorothy Graham, and Susannah Finzi. Software Inspection. Boston: Addison-Wesley, 1993.
7. Jalote, Pankaj, and M. Haragopal. Overcoming the NAH Syndrome for Inspection Deployment. Proc. of the 20th International Conference on Software Engineering. Kyoto, Japan, 19-25 Apr. 1998: 371-378.

Note

1. BDs are so named not because they are introduced as a by-product of some activity, but because they are *detected* as a by-product of an activity.

About the Author

D.T.V. Ramakrishna Rao is a senior technical architect at Infosys Technologies Limited, in Bangalore, India. He has 14 years of experience in industrial software development with a primary focus on building high-end networking systems. He has published 10 papers in networking and defect analysis. He holds a master's degree in computer science from the Indian Institute of Technology in Kanpur, India.

Infosys Technologies Limited
44 Electronics City, Hosur RD
Bangalore – 560 100
India
Phone: 91-80-41166508
Fax: 91-80-28521695
E-mail: ramakrishnadtv@infosys.com

Coming Events

May 4-6
Sea – Air – Space 2009
National Harbor, MD
www.seaairspace.org

May 4-7
SATURN 2009 Conference
Pittsburgh, PA
www.sei.cmu.edu/architecture/saturn/2009

May 4-8
Software Testing Analysis & Review
Orlando, FL
www.sqe.com/stareast

May 11-12
2009 IEEE International Conference on Technologies for Homeland Security
Waltham, MA
www.ieeehomelandsecurityconference.org

June 1-4
DoD Enterprise Architecture
St. Louis, MO
www.afei.org/brochure/9a05/

June 8-12
Better Software Conference & EXPO
Las Vegas, NV
www.sqe.com/bettersoftwareconf

June 15-19
SIGMETRICS/Performance 2009
Seattle, WA
http://conferences.sigmetrics.org/sigmetrics/2009

June 28-July 3
21st Annual FIRST Conference
Kyoto, Japan
http://conference.first.org

2010
Systems and Software Technology Conference
Salt Lake City, UT
www.sstc-online.org

A Uniform Approach for System of Systems Architecture Evaluation

Michael Gagliardi, William G. Wood, John Klein, and John Morley
Software Engineering Institute

For a large-scale system of systems (SoS), severe integration and run-time problems can arise due to inconsistencies, ambiguities, and gaps in how quality attributes (such as reliability) are addressed in the underlying systems. This is exacerbated in contexts where major system and software elements of the SoS are developed concurrently and oftentimes independently. Using a defense system scenario, this article outlines a uniform approach for capturing quality attribute requirements as augmentations to mission threads early in the development process and for analyzing SoS, system, and software architectures against these mission thread augmentations.

An SoS program, in government or industry, can suffer severe integration and run-time problems that can result in costly rework, schedule overruns, and the failure to achieve performance goals [1]. For example:

- In 2005, NASA's Demonstration of Autonomous Rendezvous Technology (DART) spacecraft collided with its target rather than achieving orbit, scuttling a $110-million mission. NASA's official investigation cited a number of contributing factors, including the failure to uncover a number of design issues prior to SoS integration [2].
- A year before the DART debacle, the Ford Motor Company killed a new supply chain system—after spending four years and about $400 million on its development—and returned to a set of "custom-written mainframe applications" for purchasing. "Poor performance" was cited as the culprit [3].
- A few years before Ford's loss, the self-developed billing and claims-processing system at Oxford Health Plans failed miserably after deployment, triggering a drop of more than $3 billion in the corporation's value. The system couldn't keep pace with the organization's needs [4].

One significant underlying cause for problems like these is a lack of attention to quality attributes (such as interoperability, sustainability, performance, and reliability[1]) early in the development life cycle, when their implications on the SoS architecture can be dealt with more easily. A recent study by the National Defense Industrial Association found that one of the top issues hindering the acquisition and successful deployment of SoS is that "insufficient systems engineering is applied early in the program life cycle, compromising the foundation for initial requirements and architecture development" [5].

® The Architecture Tradeoff Analysis Method and ATAM are registered in the U.S. Patent and Trademark Office by Carnegie Mellon University.

The typical SoS context—where major system and software elements have their own architecture documentation created by different contractors using diverse tools and notations—makes it more difficult to focus on quality attributes. In these contexts, program managers and SoS architects need a way to promote consistency, clarity, and completeness of quality attribute requirements throughout the development of the SoS.

This article describes a two-pronged, uniform approach for the early identification of quality attribute inconsistencies, ambiguities, and gaps within SoS and system architectures. Using an approach that focuses on SoS quality attribute considerations can produce benefits such as:

- Improved SoS architecture.
- Early identification of significant architectural challenges and risks.
- Better communication between the SoS, system, and software stakeholders.
- More predictable integration of component systems.
- More effective root cause analysis of problem areas.

After defining the uniform approach, we use a defense system illustration to show how this uniform approach captures considerations about reliability early in the life cycle and influences risk management throughout the SoS, system, and software development.

Uniform Approach Described

The two-pronged, uniform approach includes:

- A methodology to perform a *first pass* identification of architectural risks[2] at the SoS level, using existing mission threads that are augmented with quality attribute concerns. This is provided through a series of mission thread workshops (MTWs) and SoS architecture evaluations. The results are organized into a number of risk themes, and then individual systems are associated with these risk themes.
- Further evaluation of the problematic constituent systems can be performed using the augmented mission threads from the SoS architecture evaluations and employing an extension of the Carnegie Mellon SEI Architecture Tradeoff Analysis Method® (ATAM®) for system and software architecture evaluation (System and Software ATAM)[3].

This approach is based on successful SEI methods and techniques for addressing key quality attributes, their relationships, and trade-offs at the software architecture level. Two well-established and widely used methods are the SEI Quality Attribute Workshop (QAW) and the ATAM. The QAW helps acquirers and developers identify and characterize the key quality attributes for a system. The ATAM enables software developers and acquirers to evaluate software architecture against required quality attributes and business/mission goals before the system is actually developed. Over the past decade, many DoD programs have used the ATAM to evaluate their mission-critical software architectures.

Through this approach, architectural risks are identified early in the life cycle, promoting more efficient and effective risk management. As shown in Figure 1, the MTW takes warfare vignettes, business/mission drivers, mission threads, and SoS architecture plans as input and produces mission threads augmented with quality attribute requirements and a set of SoS architectural challenges. These augmented mission threads and architectural challenges should then be used in the development of an SoS architecture. When the SoS architecture is somewhat mature, the augmented mission threads then serve as the basis for an SoS architecture evaluation, which uncovers SoS architecture risks and points to individual systems that may pose difficulties in meeting the quality attribute concerns reflected in the mission threads. Although the emphasis in these activities is firmly on quality attributes, the evaluation often exposes functional gaps, inconsistent Concept of Operations, and process ambiguities. Then, through a System and Software ATAM, architectural risks at the system and software levels are identified.

Mission Thread Workshop

Mission threads are associated with one or more warfare vignettes. A vignette describes the overall environment, including the geography; its own force structure and mission, strategies, and tactics; and the enemy's forces, attack strategies, and tactics, including timing.

A mission thread has been defined as "sequence of activities and events beginning with an opportunity to detect a threat or element that ought to be attacked and ending with a commander's assessment of damage after an attack" [7]. Our mission thread-based example vignette is as follows:

> An enemy tank platoon is threatening a lightly protected company and comes into the field of view of an unattended ground sensor (UGS), which connects to and informs a manned ground vehicle for command and control (MGVC2). An MGVC2 identifies the enemy tanks. The MGVC2 assigns an unmanned missile launcher (UML) to engage the tank platoon. The UML engages and destroys the enemy. The MGVC2 determines that the threat has been eliminated, based on subsequent UGS signals.

The MTW is a facilitated process that brings together SoS stakeholders to both augment existing mission threads with quality attribute considerations that will shape the SoS architecture and identify SoS architectural challenges. These stakeholders include, but are not limited to, the following:

- Lead SoS architects (for the contractor and program management office [PMO]).
- Lead system and software architects (for the contractors and PMO).
- Program managers (for the contractors and PMO).
- Key representatives from integration and test, requirements, users, maintenance, installation, independent verification and validation, modeling and simulation, and other areas.

There are three main stages to the MTW: 1) preparation, 2) execution, and 3) roll-up and follow-up. In the preparation stage, the SoS program manager develops an overview presentation on the SoS mission/business drivers, and the SoS architect develops an overview presentation on the SoS architecture plans. The facilitation team meets with the SoS program manager and architect to plan the MTW, provide feedback on the two presentations, reach agreement on mission threads and types, and identify stakeholders. In this first stage, the facilitation team may need to decompose warfare vignettes into activity-oriented mission thread steps by considering:

- External actors, who may not be explicit in the vignette.
- Functionality and capability and their distribution.
- Command structure.
- Manual versus automated activation.

During the execution stage of the MTW, the SoS program manager delivers the presentation on the SoS business/mission drivers, including the business/programmatic context, the plan for development, as well as high-level functional requirements, constraints, and quality attribute requirements. The SoS architect then presents the SoS architecture plans, including: key business/programmatic requirements, key technical requirements and constraints that will drive architectural decisions, existing context diagrams, high-level SoS diagrams and descriptions, development spirals, and an integration schedule.

The bulk of the MTW execution stage is spent augmenting the mission threads with quality attribute considerations and identifying SoS architectural challenges based on stakeholder inputs and the dialogue between the stakeholders and the architects. During the augmentation, overarching quality attribute considerations—as well as step-specific quality attribute considerations for each quality attribute of the SoS—are elicited and documented for each mission thread. The architects can also describe how the planned architecture satisfies the quality attribute considerations in each step for each mission thread

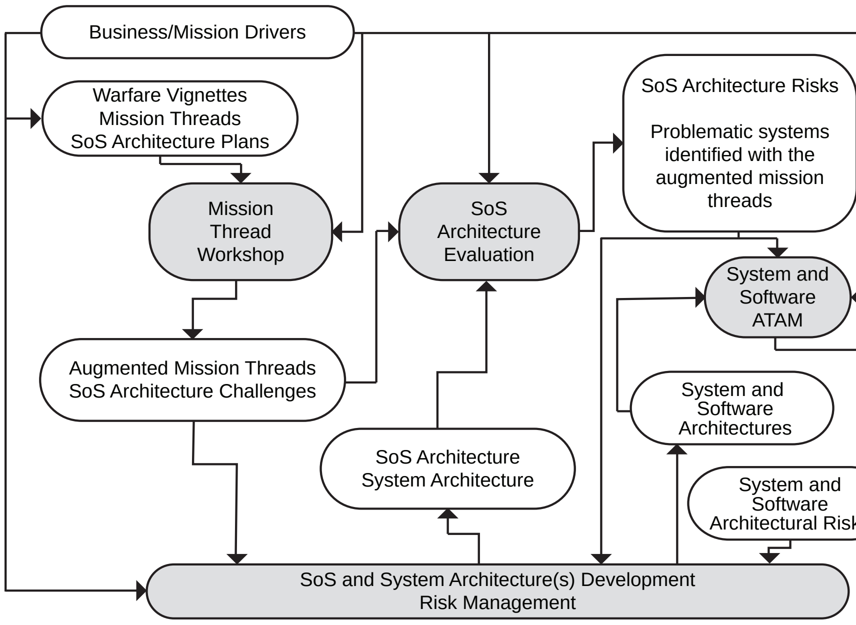

Business/Mission Drivers
Warfare Vignettes
Mission Threads
SoS Architecture Plans
Mission Thread Workshop
SoS Architecture Evaluation
SoS Architecture Risks
Problematic systems identified with the augmented mission threads
System and Software ATAM
Augmented Mission Threads
SoS Architecture Challenges
System and Software Architectures
SoS Architecture
System Architecture
System and Software Architectural Risks
SoS and System Architecture(s) Development
Risk Management

Figure 1: *A Uniform Approach for Identifying SoS Integration Problems Early*

Table 1: *Mission Thread Elements Augmented by Reliability Requirements*

Mission Thread Element	Augmentation for Reliability
An enemy tank platoon is threatening a lightly protected company and comes into the field of view of a UGS, which connects to and informs an MGVC2.	UGS-MGVC2 connection fails one second after connection; reconnects after two minutes.
An MGVC2 identifies the enemy tanks.	Takes too long (over five minutes) to de-conflict and identify the enemy tanks.
The MGVC2 assigns a UML to engage the tank platoon.	Communication transmissions to UML are missing their deadlines.
The UML engages and destroys the enemy.	Assigned UML fails to launch.
The MGVC2 determines that the threat has been eliminated from subsequent UGS signals.	Signals from UGS have ceased. Communication is lost between the MGVC2 and UGS.

selected for the workshop. Table 1 (on the previous page) shows some possible requirements for reliability expressed as augmentations on the steps in the example mission thread presented earlier.

The process used in the execution stage fosters a dialogue between architects and stakeholders regarding the issues and challenges that are captured during the workshop. In the context of our example mission thread, there could be issues such as:

- The connection with the UGS and the MGVC2 is dynamic—that is, the sensor must announce the presence of an object of interest when it senses one. More than one MGVC2 platform may be notified by the UGS, and a mechanism is needed to choose which platform will act on the data.
- The connection between the UML and the MGVC2 is also somewhat dynamic, but the MGVC2 may have an indirect connection to the UML.
- In some cases, the MGVC2 may not have the authority to engage a UML, but will have to report to a higher level command post.
- The missiles are guided in flight by the MGVC2, which may not have a direct connection.
- The tracks are created for the regional fusion engine, which may not be the MGVC2 and is not directly represented in the mission thread.

In the third and final stage, roll-up and follow-up, two reports are produced from the activity in the execution stage and answers to action items are assigned in that stage. One report includes the quality attribute considerations for each step of each mission thread selected for the MTW and overarching quality attribute considerations for each mission thread. The other report describes the SoS architectural challenges.

Experience executing QAWs (the basis of MTWs) shows that the most effective sessions bring together no more than 20 stakeholders for one to two days. This allows each MTW to tackle the augmentation of just one or two mission thread types; through a series of MTWs, the variety of mission thread types involved in an SoS are examined. At the end of the series of MTWs, an annotated summary briefing rolls up SoS architectural challenges and strengths, non-architectural issues (if any are uncovered), and recommendations.

MTWs have been used in conjunction with two large, complex DoD SoS programs. These MTWs assist architects in identifying SoS architecture challenge areas and the areas to focus their prototyping and proof-of-concept activities.

SoS Architecture Evaluation

In conjunction with the MTW, the SoS architecture evaluation provides a *first pass identification* of SoS architectural risks and quality attribute inconsistencies across the constituent systems. An SoS architecture evaluation:

- Uses outputs of the MTWs, including augmented mission threads and SoS architecture challenges.
- Incorporates the expertise of a trained evaluation team and SoS stakeholders, including the SoS and system architects.
- Probes architecture at the areas where the systems interact to identify risks.
- Organizes the individual risks into risk themes that can be comprehended (and mitigated later) by program management.
- Assesses the sufficiency of architecture documentation.
- Identifies potentially problematic systems for focused follow-on evaluations using the specific augmented mission threads.

In the SoS architecture evaluation, the SoS architect walks each augmented mission thread through the SoS architecture, describing how the SoS architecture and constituent system architectures satisfy the thread's functional and quality attribute considerations. For each step in the mission thread, the evaluation team and stakeholders probe the SoS architecture (and system architecture, if necessary) with a focus on the quality attribute augmentations and SoS challenges; risks, issues, and strengths are also identified.

At the end of each evaluation, the evaluation team delivers an outbrief that includes SoS architectural risk themes and strengths, non-architectural issues discovered, an identification of potentially problematic systems, and recommended next steps.

System and Software ATAM

As a follow-on to the SoS architecture evaluation in this approach, the System and Software ATAM keys in on the problematic systems identified. This evaluation uses the augmented mission threads to examine the system and software architecture and produces a set of software and system architectural risks that trace back to the quality attributes identified in the augmented mission threads. The System and Software ATAM is built on the format and approach of the ATAM.

Summary

Problems that do not surface until integration or deployment can have ruinous effects on the cost, schedule, and performance of SoS programs. Through the uniform approach outlined in this article, the SoS architects can use the augmented mission threads as one input for SoS architecture development. The augmented mission threads can also be used to identify any SoS architectural risks related to the quality attributes needed to accomplish the mission/business objectives early in the life cycle, promoting more efficient and effective risk management.

Table 2: *The Uniform Approach*

	MTW	SoS Architecture Evaluation	System and Software ATAM
Input	• Selected existing mission threads • Warfare vignettes • SoS architecture plans overview • SoS business/mission drivers	• Augmented mission threads • SoS architecture challenges • SoS business/mission drivers • SoS architecture • System architecture	• SoS architecture risks • Problematic systems identified with the augmented mission threads • SoS and system business/mission drivers • System architectures • Software architectures
Output	• Augmented mission threads that reflect quality attribute considerations • Architectural challenges • Issues and questions concerning the mission threads	• SoS architecture risks • Problematic systems identified with the augmented mission threads	• System architecture risks • Software architecture risks

This approach involves performing a series of MTWs and SoS architecture evaluations to identify inconsistencies, ambiguities, and gaps across the constituent systems and at the SoS level, using existing mission threads that are augmented with quality attribute concerns. It also includes further evaluation of problematic constituent systems using the augmented mission threads in a system and software version of the SEI ATAM. Table 2 summarizes the way the methods are integrated for analyzing SoS, system, and software architectures against augmented mission threads in order to expose technical risks at an early stage of development when mitigation can be done cost-effectively.

By using this uniform approach—focused on SoS quality attribute considerations early in the development life cycle—program managers and SoS architects can realize an improved SoS architecture, identification of significant architectural challenges and risks at a time when it is less costly to fix them, better communication between SoS, system, and software stakeholders, more predictable integration of component systems, and more effective root cause analysis of problem areas.◆

Notes

1. A suitable list of SoS quality attributes also includes backward compatibility, testability, and usability.
2. A risk is a potentially problematic architectural decision indicating that the system or SoS built from the architecture may not completely satisfy one or more business/mission goals.
3. The ATAM exposes architectural risks that potentially inhibit the achievement of an organization's business goals. The ATAM gets its name because it not only reveals how well an architecture satisfies particular quality goals, but it also provides insight into how those quality goals interact with each other—how they *trade off* against each other [6].

References

1. Office of the Under Secretary of Defense for Acquisition, Technology, and Logistics. "Report of the Defense Science Board Task Force on Development Test and Evaluation." May 2008 <www.acq.osd.mil/dsb/reports/2008-05-DTE.pdf>.
2. Marshall Space Flight Center. "NASA Report: Overview of the DART Mishap Investigation Results – For Public Release." 15 May 2006 <www.spaceref.com/news/viewsr.html?pid=20605>.
3. Songini, Marc L. "Ford Abandons Oracle Procurement System: Switches Back to Mainframe Apps." Computerworld Software. 23 Aug. 2004 <www.computerworld.com/software-topics/erp/story/0,10801,95404,00. html>.
4. Hammonds, Keith H., and Susan Jackson. "Behind Oxford's Billing Nightmare: How a Misconceived System Cost the Health-Care Giant Millions." Business Week. 17 Nov. 1997 <www.businessweek.com/1997/46/b3553148.htm>.
5. National Defense Industrial Association Systems Engineering Division Task Group Report. "Top Five Systems Engineering Issues Within Department of Defense and Defense Industry." July 2006.
6. Clements, Paul, Rick Kazman, and Mark Klein. Evaluating Software Architectures: Methods and Case Studies. Boston: Addison-Wesley Professional, 2001.
7. Naval Studies Board. C4ISR for Future Naval Strike Groups. Washington, D.C.: The National Academies Press, 2006.

About the Authors

Michael Gagliardi has more than 25 years experience in real-time, mission-critical software architecture and engineering activities on a variety of DoD systems. He currently works in the SEI Research, Technology, and System Solutions Program on the Architecture-Centric Engineering initiative, and is involved in the development of architecture evaluation methods for SoS architectures and system architectures.

SEI
4500 Fifth AVE
Pittsburgh, PA 15213
Phone: (412) 268-7738
Fax: (412) 268-5758
E-mail: mjg@sei.cmu.edu

William G. Wood has been a member of technical staff at the SEI for more than 22 years. During this time, he has managed a technical program and technical projects, and has provided technical support to the program development organization. Wood is currently working in software architecture with a number of clients. He has a master's degree in electrical engineering from Carnegie Mellon University, and a bachelor's degree in physics from Glasgow University, Scotland.

Phone: (412) 268-7723
Fax: (412) 268-5758
E-mail: wgw@sei.cmu.edu

John Klein is a senior technical staff member in the SEI's Research, Technology, and System Solutions Program. He does research and consulting on architecture-centric methods and tools for developing, documenting, and evaluating SoS and enterprise architectures. Klein also assesses and improves the architecture competence of individuals, teams, and organizations. Klein has more than 25 years experience in systems development including sensors and weapons, as well as collaboration systems.

Phone: (412) 268-4553
E-mail: jklein@sei.cmu.edu

John Morley is a member of the operating staff at the SEI. He has reported on model-based engineering, architecture-centric engineering, and service-oriented architecture for SEI publications and has more than 20 years experience in writing and editing scientific and technical materials. Morley recently co-wrote "Building Secure Systems Using Model-Based Engineering and Architectural Models," which appeared in the September 2008 edition of CrossTalk.

Phone: (412) 268-6599
Fax: (412) 268-5758
E-mail: jmorley@sei.cmu.edu

Static Analyzers in Software Engineering

Dr. Paul E. Black
National Institute of Standards and Technology

Static analyzers can report possible problems in code and help reinforce the good practices of developers. This article contrasts the strengths of static analyzers with testing and discusses the current state-of-the-art.

A static analyzer is a program written to analyze other programs for flaws. Such analyzers typically check source code, but there are analyzers for byte code and binaries, too. Analyzers for requirements or design are possible, but most are focused on code and binaries. At a minimum, analyzers report the location and name of a possible problem. Some analyzers have far more capabilities. They may describe the problem and possible attacks or failure modes in-depth. They may detail the data or control flow leading from the source of values involved to the *statement* where the failure may have manifested or the value is passed to another component. They may also suggest mitigations.

A *vulnerability* is any property of system requirements, design, implementation, or operation that could be accidentally triggered or intentionally exploited and result in a failure. As described in [1]: "A vulnerability is the result of one or more *weaknesses* in requirements, design, implementation, or operation." Because configuration, installation, operation, and other system components determine whether a certain code construct may lead to failure, I speak of weaknesses in the code, not vulnerabilities. To reiterate, static analyzers report weaknesses in software.

What Are Their Strengths and Limitations?

Every static analyzer has a built-in set of weaknesses to look for in code. Most have some means of adding custom rules. In contrast, testing requires test cases or input data. Testing also requires artifacts that are complete enough to be executable, possibly with supporting drivers, stubs, or simulated components. Static analysis may be performed on modules or unfinished code, although the more complete the code, the more thorough and accurate the analysis can be.

Analyzers are limited by the sophistication of the reasoning in them. For instance, some static source code analyzers do not handle function pointers and few can deal with embedded assembler code. Even if the models of the programming language, compiler, hardware, and other pieces used in execution are perfect, analyzers have the same fundamental limitation as any other logical system. They cannot solve the halting problem or undecidable problems. In practice, this need not be a serious limitation. Important code "should be so clearly correct that it confuses neither human nor tools" [2]. Although running tests is straightforward, this same challenge of analysis arises in developing tests to exercise a particular property or module.

"Most importantly, static analyzers have the potential to find rare occurrences or hidden back doors. Since they consider the code independently of any particular execution, they can enumerate all possible interactions."

New tests must be developed when new attacks or failure modes are discovered. Static analyzers have some advantage in this case. The weakness check need only be added and validated once, then the analyzer is rerun on all code. Test generators can give a similar advantage.

Most importantly, static analyzers have the potential to find rare occurrences or hidden back doors. Since they consider the code independently of any particular execution, they can enumerate all possible interactions. The number of interactions tends to increase exponentially, defying comprehensive static analysis and test execution alike. Static analysis can focus on the interaction without testing's need to re-establish initial conditions or artificially constrain the system to produce the desired interaction. Worse, black-box testing cannot realistically be expected to discover, let's say, a backdoor accessible when the user ID is "JoshuaCaleb" since there are a nearly infinite number of arbitrary strings to test.

Testing and static analysis complement each other. Testing has the advantage of possibly revealing completely unexpected failures. Embedded systems can be tested, even when it is utterly impractical to analyze any software that may be tucked away in a component.

Static Analysis' Place in Software Engineering

Static analysis is no panacea. Complex and subtle vulnerabilities can always defeat the reasoning in a static analyzer. The utter lack of an important requirement, such as auditing or encryption, cannot reasonably be deduced from only the examination of post-production artifacts. Software with no resiliency or self-monitoring is open to errors in installation or operation, but static analysis can be one of the last lines of defense against vulnerabilities.

Static analysis can be understood in a continuum from sound to heuristic. A sound analysis is 100 percent correct in its judgments. If it reports a weakness, a weakness definitely exists. If it reports that a certain construct is okay, then one is assured that a weakness is not present. In some cases, a sound analysis may not have enough information to render a good/bad judgment.

Statistical correlation is an example of heuristic analysis. For instance, an *open* is usually followed by a *close* or resources are typically locked within a critical section. Such rules may be derived automatically through machine learning of existing code. But heuristic analysis is susceptible to false alarms (false positives) or missing actual weaknesses (false negatives).

Analysis may be a combination of sound reasoning and heuristic techniques. Complete analysis of the termination conditions of every loop or possible states of all combinations of variables may be impractical, so most analyzers use algorithms that are not purely sound or purely heuristic. In addition, most analyzers are a system of analytic engines; examples are data flow, loop termination, value propagation, control flow, or property recognition.

Work from the June 2008 Static Analysis Tool Exposition [3, 4] shows that current analyzers vary widely. An analyzer may produce few false alarms for some weaknesses, but many false alarms for other weaknesses. Likewise, the rate of missed weaknesses differs greatly. Analyzers also only cover a subset of documented weaknesses [5]. Thus, the most comprehensive static analysis would result from a carefully used combination of analyzers. Other factors, such as cost and analyst support, must go into selecting the most appropriate static analyzer(s) for each situation. The Software Assurance Metrics and Tool Evaluation (SAMATE) Reference Dataset [6] has thousands of sample programs that may help such evaluation.

Static analyzers should be a key part of every software development process.◆

References

1. "Source Code Security Analysis Tool Functional Specification Version 1.0." National Institute of Standards and Technology (NIST), Special Publication 500-268. May 2007 <http://samate.nist.gov/docs/source_code_security_analysis_spec_SP500-268.pdf>.
2. Holzmann, Gerard J. "Conquering Complexity." Computer 40 (12): 111-113, Dec. 2007.
3. NIST. "Static Analysis Tool Exposition." 7 July 2008.
4. NIST. ACM SIGPLAN. Proc. of the Static Analysis Workshop. Tucson, AZ. 12 June 2008 <http://samate.nist.gov/index.php/SAW>.
5. MITRE. "Common Weakness Enumeration." 25 Nov. 2008 <http://cwe.mitre.org>
6. NIST. "NIST SAMATE Reference Dataset." Jan. 2006.

About the Author

Paul E. Black, Ph.D., has nearly 20 years of industrial experience in software for integrated circuit design and verification, assuring software quality and managing business data processing. He now works in the Software Quality Group, Information Technology Laboratory of the NIST and edits the online Dictionary of Algorithms and Data Structures. He has a doctorate in computer science from Brigham Young University and has published on topics including software testing, configuration control, networks and queuing analysis, formal methods, software verification, quantum computing, and computer forensics. Black is a member of the Association for Computing Machinery, IEEE, and the IEEE Computer Society.

NIST
100 Bureau DR Stop 8970
Gaithersburg, MD 20899-8970
Phone: (301) 975-4794
Fax: (301) 975-6097
E-mail: paul.black@nist.gov

Management's Inspection Responsibilities and Tools for Success

Roger Stewart and Lew Priven
Stewart-Priven Group

There are many pitfalls that cause software inspections to fail. This article addresses management's critical role in preventing these pitfalls to attain successful inspections. In addition to meeting their responsibilities, management needs a comprehensive computerized set of tools to support their efforts. By carrying out their responsibilities—supported by inspection-specific tools—management will be better equipped to implement sustained successful project inspections that consistently reap the benefits of lower project cost and high product quality.

For this article, an inspection is defined as a preemptive systematic peer review of work products by three to five trained individuals (e.g., stakeholders) using a well-defined and documented process. The goal of peer review is to reduce project rework cost and raise product quality and return on investment (ROI) by detecting and removing (non-trivial) defects as early as possible in the software development life cycle (SDLC) or closest to the points of defect insertion.

The objective of this article is to explore upper (middle and senior) management's software inspection responsibilities and how computerized inspection tool features can improve meeting those responsibilities for consistent inspection success.

Ten Common Inspection Pitfalls

Figure 1 identifies ten common inspection pitfalls that cause software inspections to fail. It is the responsibility of upper management to solve or prevent each pitfall. These pitfalls were the focus of our January 2008 [1] CROSSTALK article. This article is a follow-up and in the future we expect to explore inspection tool features in more depth than is discussed here.

Having an adequate SDLC infrastructure is a prerequisite for inspection success. Specifically, company culture needs an enabling SDLC infrastructure where the disciplines of planning, scheduling, data collection, monitoring, and tracking are already ingrained in the culture. The CMM® and its successor CMMI® identify these disciplines as Level 2 management skills. Inspections are dependent upon these underlying disciplines and thus require Level 2 management skills as the foundation for success. We have observed that companies typically encounter multiple inspection pitfalls, any one of which can result in not achieving the lasting benefits that inspections can provide when properly implemented.

This article addresses management's responsibilities in eliminating the pitfalls and explains how inspection tools can help make this easier to achieve.

Management's Understanding of Inspections

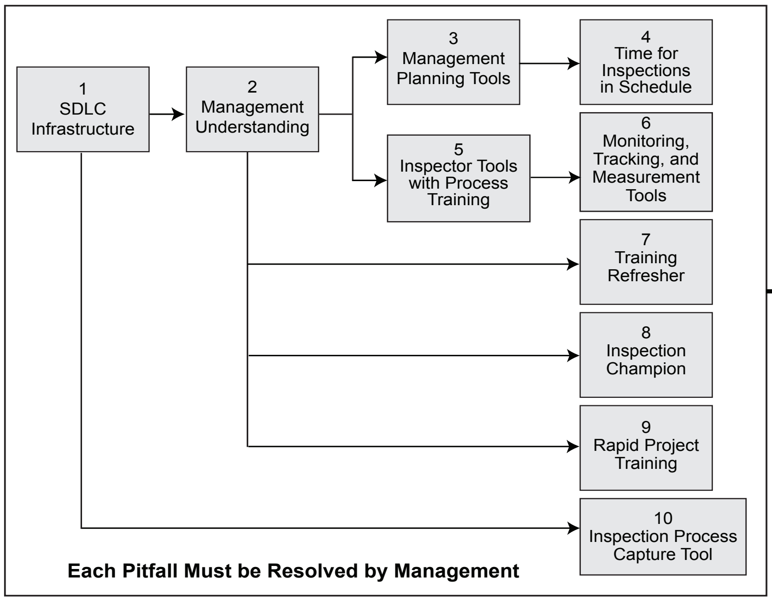
Prevention of the inspection pitfalls shown in Figure 1 requires management to first understand—and then fulfill—their inspection responsibilities. Typically when project inspections are introduced, the focus on management's critical role is either overlooked or not given the attention required. Just as inspection practitioners (inspectors) receive training on inspection process execution, management needs complementary training that focuses on identifying and achieving their inspection responsibilities.

The cost and schedule impact of inspections are primarily borne by the requirements, design, and coding areas, although these areas realize some of the reduced cost and higher-quality inspection benefits. The majority of inspection benefits are realized in testing and maintenance, requiring management to have a life-cycle view of the project. Therefore, management at all levels and across all development phases (including upper management in areas that will not use inspections), require inspection education.

Management instruction on how to manage inspections occurs prior to practitioner training and focuses on the responsibilities of management for achieving inspection success, specifically:

- Prevention of the 10 identified inspection pitfalls.
- Effective use of management support tools.

Figure 1: *Managing Software Inspection Pitfalls*


1 SDLC Infrastructure
2 Management Understanding
3 Management Planning Tools
4 Time for Inspections in Schedule
5 Inspector Tools with Process Training
6 Monitoring, Tracking, and Measurement Tools
7 Training Refresher
8 Inspection Champion
9 Rapid Project Training
10 Inspection Process Capture Tool
Each Pitfall Must be Resolved by Management
Successful Inspection Implementation

Management's Inspection Responsibilities

There are seven stages for successfully managing inspections, which are identified in Figure 2. These stages are:

- **Stage 1: Project Planning and Stage 2: Savings Estimation.** Management identifies the number of inspections required by the project, their cost, and the estimated net project savings from implementing the inspections. These two stages occur prior to conducting any project inspections.
- **Stage 3: Commitment.** Also occurring prior to conducting any project inspections, management establishes and incorporates inspection planning and control

® CMM and CMMI are registered in the U.S. Patent and Trademark Office by Carnegie Mellon University.

procedures into the project's plans. Elements of the commitment stage are described later in the article.

- **Stage 4: Execution.** Management's principal responsibility in this stage is to provide inspection tools that facilitate correct and consistent (repeatable) execution of inspections by teams, organizations, and locations supporting their project. Additionally, these inspection tools should automate the collection of inspection data for later monitoring, tracking, and measurement of inspections results.
- **Stage 5: Monitoring.** Management assesses the interim inspection process conformance and results. If either appears questionable, then consultation with the inspection team leader may reveal that a partial or full reinspection is needed before further time is invested investigating and fixing the potential problems that have been uncovered. This stage also provides early identification of potential defect-prone areas and the need for improvement to inspection materials, team selection, and inspection process adherence.
- **Stage 6: Tracking.** After each inspection exit, individual inspection results are collected and consolidated into multiple inspection totals used for tracking to-date project savings against the stage 2 savings estimate. Tracking also includes the ongoing trend analysis of defect reason and origin metrics collected during inspections, which are used for future defect prevention improvements to development processes.
- **Stage 7: Measurement.** This stage occurs throughout the development life cycle, providing for evaluation of early defect removal by inspections, defect removal by subsequent testing, and any other means of defect removal (e.g., pretest automated code analysis tools). This stage also provides for the evaluation of the effectiveness of the inspection implementation for pre-test defect removal at or close to the points of insertion. It is also a means to assess progress toward the defect removal goals recorded in the project's quality plan (described later).

These stages should not be confused with the seven inspection steps typically associated with performing inspections [2], which occur during the execution and monitoring stages of management's inspection responsibilities (see Figure 2).

With this understanding of management's responsibilities, let's take a detailed look at how a set of inspection management tools might be used.

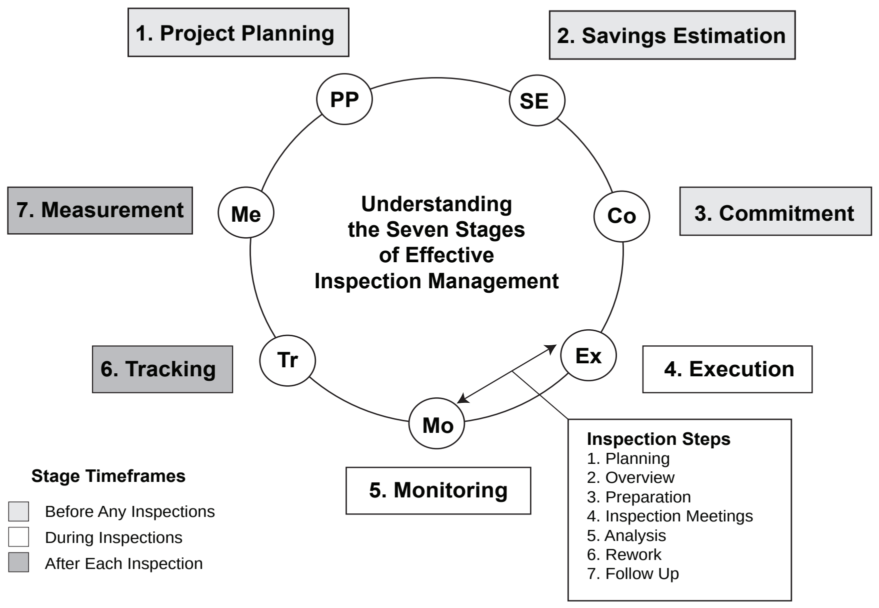


Figure 2: *Stages of Management's Inspection Responsibility*

Management Inspection Tools

As previously stated, without management tools designed for each inspection stage and integrated with each other to build upon inspection information collected in earlier stages, management will be challenged to meet their inspection responsibilities and achieve project success.

Figure 3 identifies what a set of computerized inspection tools might consist of. The tools shown are grouped according to the four timeframes where they would be used:

- Before Any Inspections (during project planning).
- During Inspections (execution and monitoring).
- After Each Inspection (tracking).
- Throughout Development and Testing (measurement).

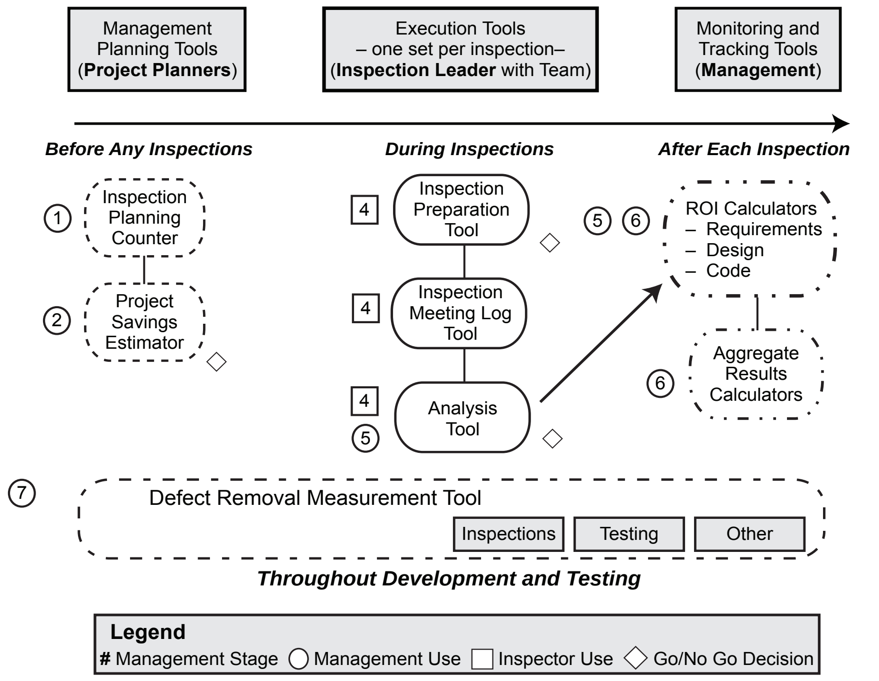
The legend in Figure 3 shows the association of each tool with the inspection stage they apply to (Figure 2), whether the tool is for use by management or by inspection practitioners, and where the three tool-aided decision points are for proceeding with an inspection.

Inspection Tool Use

Before Any Inspections

During the first stage (project planning), a planning counter tool should be used to compute the number of inspections to be conducted during a project, the estimated

Figure 3: *Computerized Inspection Tool Use*


Management Planning Tools (Project Planners)
Execution Tools – one set per inspection– (Inspection Leader with Team)
Monitoring and Tracking Tools (Management)
Before Any Inspections
During Inspections
After Each Inspection
1 Inspection Planning Counter
2 Project Savings Estimator
4 Inspection Preparation Tool
4 Inspection Meeting Log Tool
4 5 Analysis Tool
5 6 ROI Calculators – Requirements – Design – Code
6 Aggregate Results Calculators
7 Defect Removal Measurement Tool
Inspections
Testing
Other
Throughout Development and Testing
Legend
# Management Stage ○ Management Use □ Inspector Use ◇ Go/No Go Decision

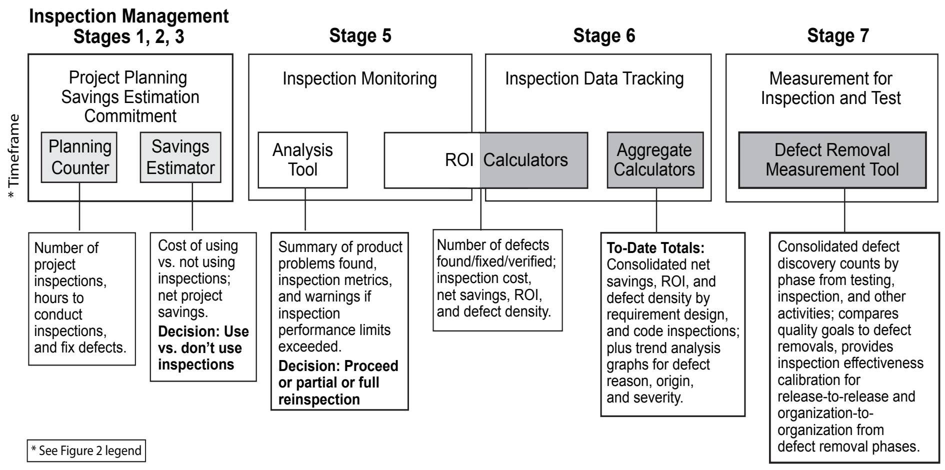

Inspection Management Stages 1, 2, 3
Stage 5
Stage 6
Stage 7
* Timeframe
Project Planning Savings Estimation Commitment
Planning Counter
Savings Estimator
Inspection Monitoring
Analysis Tool
Inspection Data Tracking
ROI Calculators
Aggregate Calculators
Measurement for Inspection and Test
Defect Removal Measurement Tool
Number of project inspections, hours to conduct inspections, and fix defects.
Cost of using vs. not using inspections; net project savings. Decision: Use vs. don't use inspections
Summary of product problems found, inspection metrics, and warnings if inspection performance limits exceeded. Decision: Proceed or partial or full reinspection
Number of defects found/fixed/verified; inspection cost, net savings, ROI, and defect density.
To-Date Totals: Consolidated net savings, ROI, and defect density by requirement design, and code inspections; plus trend analysis graphs for defect reason, origin, and severity.
Consolidated defect discovery counts by phase from testing, inspection, and other activities; compares quality goals to defect removals, provides inspection effectiveness calibration for release-to-release and organization-to-organization from defect removal phases.
* See Figure 2 legend

Figure 4: *Overview of Management's Inspection Tool Features*

number of defects to be removed, and the projected cost for conducting project inspections to find and fix those defects.

During the savings estimation stage, a savings estimator tool would then use the planning counter tool output along with historical data and industry data to estimate the net savings (or cost) the project would realize (from the inspections and the resulting reduced rework and testing). Using this data, management can now make an informed decision whether to employ inspections.

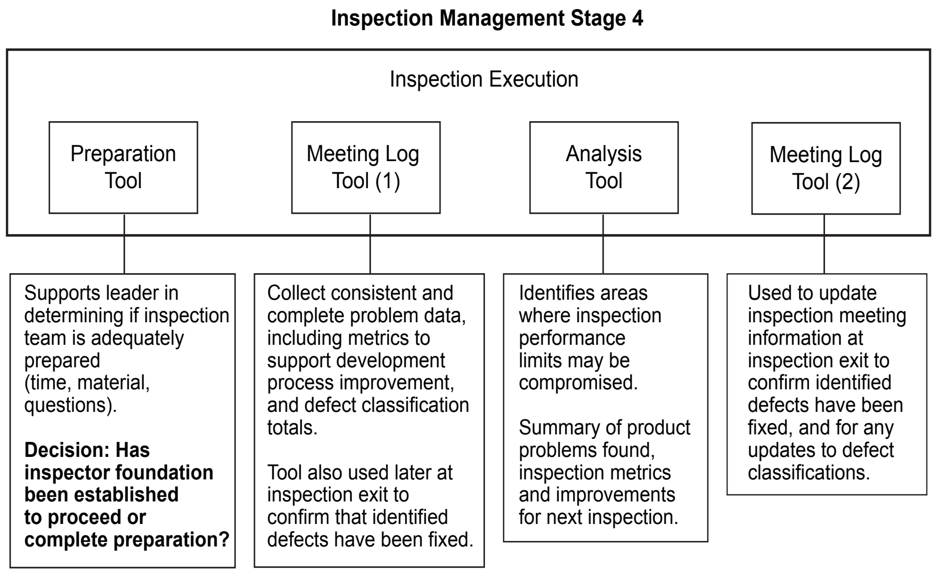
Our experience is that few project managers will really commit to inspections without knowing the net savings benefit from implementing inspections. Few organizations (if any) know how to compute their true inspection cost and, more importantly, accurately estimate their net project savings from implementing inspections as a primary means of raising product quality and lowering both development and maintenance costs. Using both planning counter and savings estimator tools can provide management with the ability to compute the net project savings estimate.

Computerized inspection tools are a critical aid to management for both of the first two stages. Furthermore, if these tools are parameterized to accept historical enterprise data and possible future development and testing profiles, then management has the additional flexibility to examine what-if scenarios that lead to making an informed decision for their project to use, or not use, inspections. This information can also be used to sell or defend their decision, as needed.

The commitment stage—performed prior to conducting any project inspections—includes the following responsibilities:

- Conveying a strong net-savings mindset (as opposed to a cost focus) so often missing from projects trying to do inspections.
- Securing buy-in from upper management across all project areas (e.g., requirements, design, code, and test).

Figure 5: *Summary of Practitioner Inspection Tools*


Inspection Management Stage 4
Inspection Execution
Preparation Tool
Meeting Log Tool (1)
Analysis Tool
Meeting Log Tool (2)
Supports leader in determining if inspection team is adequately prepared (time, material, questions).
Decision: Has inspector foundation been established to proceed or complete preparation?
Collect consistent and complete problem data, including metrics to support development process improvement, and defect classification totals.
Tool also used later at inspection exit to confirm that identified defects have been fixed.
Identifies areas where inspection performance limits may be compromised.
Summary of product problems found, inspection metrics and improvements for next inspection.
Used to update inspection meeting information at inspection exit to confirm identified defects have been fixed, and for any updates to defect classifications.

The estimated net savings output of a savings estimator tool is ideal for accomplishing these responsibilities.

The other elements of inspection commitment responsibilities can be accomplished without additional inspection tools by:

- Establishing a quality plan with quantifiable defect removal goals for each development phase (requirement definition through testing and customer use).
- Developing project plans that contain individual inspection schedules.
- Developing inspection budgets and/or setting up cost accounts for tracking inspection cost.
- Scheduling upper management inspection awareness classes.
- Securing rapid practitioner training that doesn't have delays between multiple practitioner classes (a characteristic of many company training departments).
- Employing an inspection methodology (see [1]) that resolves all inspection pitfalls in addition to accomplishing both management and practitioner inspection training.

Another aspect of management commitment is to provide an inspection environment that facilitates inspection practitioners in attaining the full quality and cost savings benefits of inspections. To accomplish this, an inspection tool set should be obtained for practitioners that:

- Assists inspector adherence to both the inspection process [2] and performance limits that characterize successful inspections.
- Provides warnings where proceeding to the next inspection step could be risky.
- Collects complete and consistent inspection data for inspection monitoring and savings tracking.
- Provides a snapshot report for monitoring inspection process adherence and interim pre-fix inspection results.
- Provides defect metrics for the post-inspection pursuit of development process improvements for future defect prevention.

During Inspections

Managers of inspection practitioners do not attend inspection meetings, but management needs the capability to monitor interim results at the conclusion of inspection meetings (step 4 in Figure 2). This capability must be uniform and repeatable regardless of what team, organization, or location is conducting project inspections.

Ideally, a one-page snapshot from the tool output of the inspection analysis step will provide management with the data needed to meet their inspection monitoring responsibility. This would include identifying where inspection performance thresholds have been exceeded. This will assist management in determining whether a partial or full reinspection is needed. Inspection performance thresholds might include the number of lines inspected, inspection meeting length, preparation rate, inspection rate, number of inspectors, number of inspectors with adequate domain knowledge and language knowledge, adequacy of inspection materials to meet inspection entry criteria, and adequacy in individual inspector preparation. While some threshold violations may be suspected or possibly known prior to an inspection meeting (e.g., preparation rate), they should also be included on a post-inspection meeting report for management to get the complete picture on whether to allow the inspection to move forward into rework (step 6 in Figure 2) or whether some form of reinspection is needed.

After Each Inspection

The tracking stage occurs immediately upon completing each inspection (i.e., exiting inspection step 7 in Figure 2). Inspection results are consolidated, typically by inspection type (e.g., requirements, design, code) to provide management up-to-date insight into project net-savings and ROI.

Other inspection metrics like defect density and defect fix counts can provide an early warning of defect-prone areas and development process weak spots. Consolidating post-inspection data also provides insight into the level of inspection participation by each development area (e.g., requirements) across the project, which can be compared with the planned number of inspections computed during the project inspection planning stage.

Throughout Development and Testing

Finally, there needs to be a means to calibrate whether a project's inspection objective to find and fix defects close to their point of insertion is being achieved. Or, to present this idea as a question: Is the project's inspection implementation improving its early defect removal ability from release-to-release, or has defect removal degraded and (in turn) bogged down testing?

To address this question, quality goals for early defect removal should be established (during planning) that are similar to the typical profile of defect insertion, sometimes referred to as "defect potentials" [3]. A TRW study found that 80 percent of product defects are inserted prior to coding (52 percent during requirements and 28 percent during design) [4]. The project goals for defect removal should strive to remove defects at the same rate as they are being inserted into a product (inspections were designed for this). Defect removal goals should be established for each development phase and recorded in the project's quality plan.

To measure the effectiveness of inspections, actual defect removal data collected from each development phase should be compared to the quality plan goals for early defect removal. To accomplish this, a defect removal measurement tool should be employed that is designed to collect, on a phase-by-phase basis, the:

- Current or past defect removal profile.
- Defect insertion (potentials) profile.
- Project defect removal goals.
- Actual defect removal metrics from inspections, testing, and any other activities.

The defect removal measurement tool could then compare the phase-by-phase defect removal goals versus actual defect removals so management can track release-to-release goal attainment or degradation. The actual defect removal phase profile (from a release) is then used to help set the defect removal goals for the next release.

Inspection Tool Features

Figure 4 provides an overview of the key features previously discussed that management inspection tools should possess.

Summary

Inspections can live up to their potential and be embraced by the development community if:

- Inspections are integral to a well-defined SDLC infrastructure, supported by upper management in each phase of development.
- Computerized management tools are available to assist in planning project inspections and estimating net project savings before commitment to inspections.
- Computerized management tools are available for monitoring inspection process conformance, tracking resulting benefits for multiple inspections, and measuring (calibrating) the actual defect removal effectiveness of inspections.
- Project management and inspection practitioners are provided with training tailored to their unique inspection responsibilities.
- Computerized practitioner tools are available to guide inspection teams for consistent, correct, and repeatable inspection execution (as shown in Figure 5), and to be the basis for management monitoring, tracking, and measurement phase responsibilities.◆

References

1. Priven, Lew, and Roger Stewart. "How to Avoid Software Inspection Failure and Achieve Ongoing Benefits." CrossTalk Jan. 2008.
2. Software Engineering Standards Committee of the IEEE Computer Society. "IEEE Standard for Software Reviews, Section 6. Inspections." IEEE Std. 1028-1997, 1997.
3. Jones, Capers. "Measuring Defect Potentials and Defect Removal Efficiency." CrossTalk June 2008.
4. McGraw, Gary. "Making Essential Software Work." Cigital, Inc. Mar. 2003.

About the Authors

Roger Stewart is co-founder and a managing director of the Stewart-Priven Group. He is an experienced lead systems engineer and program manager in both government and commercial systems, including systems engineering, software development, system integration, system testing, and process improvement. Stewart has a bachelor's degree in mathematics from Cortland University.

Lew Priven is co-founder and a managing director of the Stewart-Priven Group. He is an experienced executive with a management and technical background in system and software development, software quality training, management development training, and human resource management. Priven has a master's degree in management from Rensselaer Polytechnic Institute and a bachelor's degree in electrical engineering from Tufts University.

The Stewart-Priven Group
7962 Old Georgetown RD STE B
Bethesda, MD 20814
Phone: (865) 458-6685
Fax: (865) 458-9139
E-mail: spgroup@charter.net

WEB SITES

Software Program Managers Network (SPMN) Critical Software Practices

www.spmn.com/16CSP.html

The SPMN developed Critical Software Practices to specifically address underlying cost and schedule drivers that have caused many software-intensive systems to be delivered over budget, behind schedule, and with significant performance shortfalls. These practices are the starting point for structuring and deploying an effective process for managing large-scale software development and maintenance. Together, they constitute a set of high-leverage disciplines that are focused on improving a project's bottom line. They may be tailored to the particular culture, environment, and program phase.

Making the Business Case for Software Assurance Workshop

www.sei.cmu.edu/community/BCW_Proceedings.pdf

Last September, many of the software industry's best and brightest met at Carnegie Mellon University for SEI's Making the Business Case for Software Assurance Workshop. The goal was to have researchers and practitioners from the fields of software engineering, system engineering, software security, and software assurance exchange ideas and their experiences in support of a business case for software assurance. This comprehensive document captures the topics presented and discussed, including measurement, process and decision making, legal issues, globalization, risk, and organizational development. Also included is the keynote address from longtime CROSSTALK contributor Joe Jarzombek.

Workshop on the Evaluation of Software Defect Detection Tools

www.cs.umd.edu/~pugh/BugWorkshop05/

Learn more about the topics discussed—and read the papers presented—at this gathering of researchers and developers of software defect detection tools. Topics included defect types, defect categorization, defect prioritization, false positives, static and dynamic analysis techniques, defect database mining, defect tool interface issues, and post-deployment defect detection.

Source Code Security Analysis Tool

https://samate.nist.gov/docs/source_code_security_analysis_tool_spec_01_29_07.pdf

This document—from the National Institute of Standards and Technology in conjunction with the Department of Homeland Security—specifies the functional behavior of one class of software assurance tool: the source code security analyzer. Because the majority of software security weaknesses today are introduced at the implementation phase, a specification that defines a "baseline" source code security analysis tool capability can help software professionals select a tool that will meet their software security assurance needs.

The Software Practices Lab

www.cs.ubc.ca/labs/spl

Based at the University of British Columbia, the Software Practices Lab is a group of researchers with the shared goals of improving software development practices, making real-world software development more productive, and producing better systems. Projects and papers on their Web site explore all parts of the software life cycle and apply a wide range of techniques, including programming environments, meta programming, scalable source analysis, software evolution support, programming language design and implementation, and case study development.

Software Inspections

www.methodsandtools.com/archive/archive.php?id=29

Software inspection has been around for more than 35 years, but some are still not convinced of its value. In this article from *Methods and Tools* magazine, Ron Radice of Software Technology Transition shares his experiences with successful inspections and explains why they are an invaluable tool. Radice explores the value of software inspections independent of software development standards, the role of programmers in causing defects, the objective of reducing Cost of Quality during inspections, inspection effectiveness and efficiency, roles and tasks during an inspection, minimally staffed inspection teams, and what lessons can be learned from both successful and failed inspection experiences. Radice also answers questions regarding whether software inspection can replace testing and if software inspection will become obsolete.

Effective Software Sizing

www.pmforum.org/library/papers/2007/PDFs/Galorath-407.pdf

As most software professionals know, software size directly relates to development effort: The more accurate your size estimate is, the more accurate cost and schedule estimates will be. But despite all the current IT monitoring of schedules and budgets, project overruns are alarming. Past CROSSTALK contributor Dan Galorath examines both the problems in estimating software size—and the solutions. What are the keys to achieving accurate size estimates? What are you trying to estimate? How will you measure your project? What is the difference between total size and effective size? Since single point values don't tell the whole story, how do you express uncertainty? These questions are asked and answered by Galorath, who also examines the estimation techniques of expert judgment, Delphi analysis, analogy, and database comparison, and provides a case study showing these sizing techniques in action.

The National Centers for Systems of Systems Engineering (NCSOSE)

www.eng.odu.edu/ncsose

NCSOSE is a research center established to lead and facilitate academia, government, and industrial organizations in resolving problems, developing technologies, and directing research relating to major issues in the field of systems of systems (SoS) engineering. Engineering of these new higher-order metasystems must be capable of maximizing the SoS performance, as opposed to individual performance of subordinate subsystems or peer systems. NCSOSE is dedicated to the realization of the theory, methods, and practice necessary for success in the emerging environment for the design, analysis, operation, maintenance, and transformation of SoS.

The Evolution of Software Size: A Search for Value©

Arlene F. Minkiewicz
PRICE Systems, LLC

Software size measurement continues to be a contentious issue in the software engineering community. This article reviews software sizing methodologies employed through the years, focusing on their uses and misuses. It covers the journey the software community has traversed in the quest for finding the right way to assign value to software solutions, highlighting the detours and missteps along the way. Readers will gain a fresh perspective on software size, what it really means, and what they can and cannot learn from history.

When I first started programming, it never occurred to me to think about the size of the software I was developing. This was true for several reasons. First of all, when I first learned to program, software had a tactile quality through the deck of punched cards required to run a program. If I wanted to size the software, there was something I could touch, feel, or eyeball to get a sense of how much there was. Secondly, I had no real reason to care how much code I was writing; I just kept writing until I got the desired results and then moved on to the next challenge. Finally, as an engineering student, I was expected to learn how to program but was never taught to appreciate the fact that developing software was an engineering discipline. The idea of size being a characteristic of software was foreign to me—what did it really mean and what was the context? And why would anyone care?

Now, 25 years later, if you Google the phrase *software size* you will get more than 100,000 hits. Clearly, there is a reason to care about software size and there are lots of people out there worrying about it. And still, I am left to wonder: What does it really mean and what is the context? And why does anyone care?

It turns out that there are several very good reasons for wanting to measure software size. Software size can be an important component of a productivity computation, a cost or effort estimate, or a quality analysis. More importantly, a good software size measure could conceivably lead to a better understanding of the value being delivered by a software application. The problem is that there is no agreement among professionals as to the right units for measuring software size or the right way to measure within selected units.

This article examines the various approaches used to measure software size as the discipline of software engineering evolved throughout the last 25 years. It focuses on reasons why these approaches were attempted, the technological or human factors that were in play, and the degree of success achieved in the use of each approach. Finally, it addresses some of the reasons why the software engineering community is still searching for the right way to measure software size.

Lines of Code

As software development moved out of the lab and into the real world, it quickly became obvious that the ability to measure productivity and quality would be useful and necessary. The lines of code (LOC) measure—including source LOC (SLOC), thousands of LOC, and thousands of SLOC—is a count of the number of machine instructions developed. It was the first measure applied to software, with its first documented use by R.W. Wolverton in his attempt to formally measure software development productivity [1].

"Now, 25 years later, if you Google the phrase software size you will get more than 100,000 hits. Clearly, there is a reason to care about software size and there are lots of people out there worrying about it."

In the '70s, the LOC measure seemed like a pretty good device. Programming languages were simple and a fairly compelling argument could be made about the equivalence among LOC. Besides, it was the only measure in town.

In the late '70s, RCA introduced the first commercially available software cost estimation tool, which used SLOC converted to machine instructions as the size measure for software items being estimated. In the '80s, Barry Boehm's COCOMO was introduced, also using SLOC as the size measure of choice. As other cost models followed, they too used LOC measures to quantify the amount of functionality being delivered. It is important to note that while all of these models used SLOC as a primary cost driver, there are many other factors that influence the cost of software development as well. These software cost estimation models also need to gather information about factors such as the complexity of the software being estimated, the experience and capability of the software development team, expected reuse, the overall productivity of the development organization, constraints, and so forth. The quantification of these factors is applied to the software size to determine estimates of cost and effort.

I believe that SLOC will go down in the annals of engineering history as the most maligned measure of all time. There are many areas where criticism of SLOC as a software size measure is justified. SLOC counts are, by their nature, very dependent on programming language. You can get more functionality with a line of Visual C++ than you can with a line of FORTRAN, which is more than you get with a line of Assembly Language. This does make using SLOC as a basis for a productivity or quality comparison among different programming languages a bad idea.

Capers Jones has gone so far as to label such comparisons "professional malpractice" [2].

Concerns also surround the consistency of SLOC counts, even within the same programming language. There are several distinct methods for counting LOC. Counting physical LOC involves counting each line of code written while logical lines involve counting the lines that represent a single complete thought to the compiler. In many programming languages, spaces are inconsequential; because of this, the differences between physical and logical lines can be significant. Add to this the fact that even within each of these methods, there are questions as to how to deal with blanks, comments, and non-executable statements (such as data declarations). Programmer style also influences the number of LOC written as there are multiple ways a programmer may decide to solve a problem with the same language.

Additionally, if SLOC is the only characteristic of a software program that is measured, productivity and quality studies will overlook many important factors. Other important characteristics include the amount of reuse, the inherent difficulty of solving a particular problem, and environmental factors that model the approaches and practices of an organization. All of these things influence the productivity of a project.

In general, it is fair to say that SLOC measurement, considered in a vacuum, is a poor way to measure the value that is delivered to the end user of the software. It does continue to be a popular measure for software cost and effort estimation. Even as other metrics have emerged that are considered *better* by much of the software engineering community, many of the popular methodologies used for estimation rely on SLOC; many go so far as to convert the *better* measures into SLOC before actually performing estimates.

There are several likely factors as to why the SLOC method continues to be used despite its many limitations. Many of the organizations that care about software measurement have historical databases based on SLOC measures. So, although it is a valid argument that SLOC are impossible to estimate at the requirements phase of a project, it is not hard to understand why so many organizations find that they can do it successfully within their own product space. They have calibrated their processes and understanding around this and have met significant success using the SLOC method for estimation and measurement within the context of their projects and practices. Another important consideration is the fact that once an organization has agreed on measurement rules for SLOC, counting can be automated so that completed projects can be measured with minimal time and effort and without subjectivity.

Function Points

In 1979, Allan J. Albrecht introduced function points, which are used to quantify the amount of business functionality an information system delivers to its users [3]. Where SLOC represents something tangible that may or may not relate directly to value, function points attempt to measure the intangible of end user value. Function point counts look at the five basic things that are required for a user to get value out of software: Input, Outputs, Enquiries, Internal Data Stores, and External Data Stores. A function point count looks at the number and complexity of each of these components in order to determine the *amount* of end user functionality delivered. Function points create a context for software measurement based on the software's business value.

"Where SLOC represents something tangible that may or may not relate directly to value, function points attempt to measure the intangible of end user value."

Function points also offer a way to measure productivity that is independent of technology and environmental factors. It doesn't matter what programming language is being used or how mature the technology is, it doesn't matter how verbose or terse the programmers are, it doesn't matter what hardware platform is used—100 function points is 100 function points. This provides businesses a way of looking at various software development projects and assessing the productivity of their processes.

While I would be remiss not to acknowledge the great contribution that Albrecht made to the software engineering community with the introduction of function points, I would be equally remiss to stop the story here. Function points are not the answer to all software measurement woes, as they come with their own set of limitations.

Albrecht developed function points to address a specific problem within his organization, IBM. They, like many businesses that developed software, were concerned with the problem of runaway software projects and wanted to get a better handle on their software development processes. According to Tom DeMarco, "you can't manage what you can't measure" [4]. Function points related very closely to the types of business applications that IBM was developing at the time, proving to be a far superior measure of business value than SLOC; function points can be much better for an organization that develops these types of systems to use for productivity comparison studies.

It's fair to say that function points caught on like wildfire in the software engineering community. Many new and successful businesses grew around helping software development organizations use function points to improve their measurement and quality programs, especially for commercial IT software developments. Two problems grew out of the introduction of function points. The first was that the fervor to jettison the much-maligned SLOC measures caused many to embrace function points for all types of systems, many not well-suited to function points. The second was that many tried to use function points as a panacea for all measurement problems.

Function points work best for data-intensive systems where data flows, input screens, output reports, and database inquiries dominate. As the industry tried to use function points to measure the business value of real-time systems, command and control systems, or other systems with several internal logical functions, they consistently under-represented the value that these systems delivered. It turns out that information about inputs, outputs, and data stores is not adequate for determining the value of software that has a lot going on behind the scenes. In 1986, Software Productivity Research developed feature points to try to address this shortcoming with function points. The feature point definition added algorithms to the entities that are counted and weighted. Mark II function points were introduced by Charles Symons and Boeing introduced three-dimensional function points. The Common Software Measurement International Consortium's (COSMIC) full function points were unveiled in the late '90s and became ISO-certified in 2003. COSMIC function points provide multiple measurement views, one from the perspective of the user and one from the perspective of the developer. All of these alternate methods were intended to address one or more of the weaknesses or limitations of Albrecht's function points—now commonly referred to as International Function Point Users Group (or IFPUG) function points. The industry loved the idea of having a point system to define value, but, as with SLOC, the industry could not agree on the best way to measure points.

Despite the limitations and obstacles, the industry finally had a better way to measure productivity for software development projects. And, if you can use it to measure productivity, it certainly can be used to estimate new projects as well. If your organization knows how many days it takes to build a function point, planning projects into the future should be a breeze. But a crazy thing happened when organizations started using function points to estimate projects: They discovered that things other than business value drove project costs. While function points were good for measuring organizational productivity, they weren't really fitting the bill for estimating cost and effort. The value adjustment factor (VAF) was added to the definition of a function point in a rather weak attempt to address this limitation. VAF takes into account general systems characteristics such as the amount of online processing, performance requirements, installation ease, and reusability. It then uses those characteristics to adjust a function point count based solely on functional user requirements. With the VAF, the function point community managed to stray from business value while adding very limited additional ability to accurately predict development costs. Estimating costs using value-adjusted function points became its own form of professional malpractice.

Function points, in their many variations, offer the software engineering community a better window into business value, although the existence of many definitions does not lead to the cross-cultural comparisons of the productivity desired. They still present a good tool for organizations that develop comparable software products to use for both benchmarking and determining best practices. There are, of course, additional limitations with function points. Although well-documented rules exist for counting function points, there is still subjectivity in the interpretation of these rules. Furthermore, the process of counting function points has yet to be effectively automated; the manual process is time-consuming and requires professional certification.

Other Size Measurements

Other sizing measures have been introduced over the years as well. In the '80s, as object-oriented (OO) design and development gained popularity, there was a flurry of activity to develop software measurements related specifically to artifacts that came from OO designs. These measures made it possible to perform productivity studies across similar projects. Little was done, however, to relate these artifacts to the value that the software delivers, making these studies less applicable outside of a specific application domain. Additionally, because a design was required in order to assess these artifacts, the measures were not particularly suited to estimation. OO metrics never really caught on in a widespread fashion, although there are pockets within the community that have found OO measures they are happy with and can use effectively.

"But a crazy thing happened when organizations started using function points to estimate projects: They discovered that things other than business value drove project costs."

There is a measure which grew out of object orientation that shows some promise in the representation of business value. Use case points were introduced in 1993 by Gustav Karner (see [5]), with use cases being introduced by Ivar Jacobson in the mid-80s [6]. Use cases provide a language for describing the requirements of a software system in a way that facilitates communication between developers and the eventual users of the system. Each use case describes a typical interaction that may occur between a user (human operator or other software system) and the software. The focus is on the functions that a user may want to perform or have performed, rather than on how the software will actually perform those functions. Use case points count and classify the actors in the use case and the transactions that are required to make the use case happen. Use case points describe the functionality being delivered rather than the way this functionality is implemented; in other words, they describe business value. As with function points, there are still technical and implementation details that must be addressed on top of business value when used for estimation. Unlike function points, the use case points can cover a wider spectrum of application types. The problem with utilizing use cases is their lack of standardization across the industry and even across organizations. An organization that has a well-defined process for defining use cases could successfully use them for productivity tracking and effort estimation.

Agile software development practices are adding additional options for measurement of the output and productivity of software projects. Agile development offers a relatively new paradigm for the successful production of software solutions. The tenets of Agile include very short, well-contained iterations of software development that can be carefully measured with respect to the output of business value. Measures of story points, acceptance tests passed, and unit tests created and/or passed replace more traditional measures with values that speak more directly to the business value added in each iteration. Story point measures focus on functionality that provides end-to-end business value. They are defined within the software development group and are used by the group to estimate effort and to measure the productivity of successive iterations. Over time, with discipline, these groups become proficient at assigning story points to the software features they are asked to develop. Test cases developed and/or passed measure the quality dimension of business value. Agile measures such as story points and tests passed, while having little value outside of a specific software development group for benchmarking or comparison studies, offer a great deal of external value for communicating productivity and quality and provide an excellent tool for negotiating features with management.

Future of Software Sizing

The software industry has struggled during the last 25 years to find the right way to assess the productivity and quality of software development projects. The entire industry continues searching for solutions because high-quality assessment methods are necessary for proper project planning and execution. It is important to understand organizationally how productive our software development ventures are. Organizations hoping to improve software processes also measure in order to benchmark their organization against others considered *best in breed*. The formula for productivity is output divided by effort. Our struggle has centered on finding the right units to describe output.

Clearly, LOC are a very tangible output of the software development process. Just as clearly, they are unsuitable to measure productivity except in very tightly constrained environments because there is no clear relationship between a SLOC count and the amount and complexity of *features* delivered to the end user. Function points, feature points, and all the other derivations of this concept are not real and thus cannot be considered *output* of the software development process. They do, however, supply, in many cases, a quantification of features being delivered to the user. As such, they have promise, within a defined scope, as a measure for productivity across organizations. On their own, they are not sufficient to estimate future software development efforts because they don't measure non-functional requirements that sometimes have significant impacts on the amount of effort required in software development. Additional units of measure have been introduced and have gained some success within pockets of the community, but nothing has managed to achieve widespread popularity.

The software community continues to struggle with measurement issues because they continue to seek the *silver bullet* solution. Every measurement exercise needs to be conducted within a certain context and the temptation to apply one unit of measurement to answer all problems should be avoided. In a perfect world, it would be possible to establish a one-to-one correspondence between the effort associated with a software development project and the business value delivered by that project; in the real world, however, there are other factors that come into play. What seems clear is that with discipline, rigor, and well-defined practices, organizations can be successful using any unit for software size for internal project planning and productivity studies.

So far, we have failed to identify a universally applicable measure for size. The scope of a software project has multiple dimensions. The amount of user functionality is an important dimension but, if viewed alone, it has limited value outside of a very narrow context. External benchmarking and productivity studies need to be performed within stratified categorizations of feature complexity and non-functional requirements.

While there is still no answer to the question of what's the best way to measure the output of a software development project, technology appears to be leading us in a positive direction. You can't open your inbox without finding a few spam e-mails talking about service-oriented architecture (SOA), cloud computing, or some other configuration that separates the implementation of business rules from the implementation of the logistics necessary to deliver these business rules—or, in other words, configurations that separate IT-type functionality from business-type functionality. While still not a silver bullet, organizations that are truly able to achieve service orientation put themselves in a position to both measure the business value of software projects and predict the cost of delivery of future business value using the same units of measurement. This unit of measure, however, may still require definition; if a way can be found to quantify services, that may lead to a better solution to the software measurement quandary. As SOA and related technologies become more widespread (if they do actually become more widespread), this is definitely an area for further research.

The software engineering community should be commended for efforts in size measurements. There have been significant strides during the last quarter century in an effort to evolve measurement practices. There is a continued pursuit of a better measure to describe the output and productivity of software development projects. Simultaneously, the software engineering community is attempting to bridge the gap between IT and the business by working towards a business value-based language to describe our software.◆

References

1. Wolverton, R.W. "The Cost of Developing Large-Scale Software." IEEE Transactions on Computers Vol. C-23, No. 6: 615-636, June 1974.
2. Jones, Capers. "Measuring Defect Potentials and Defect Removal Efficiency." CROSSTALK June 2008.
3. Albrecht, Allan J. Measuring Application Development Productivity. Proc. of the Joint SHARE, GUIDE, and IBM Application Development Symposium. 14-17 Oct., Monterey, CA. IBM Corporation, 1979.
4. DeMarco, Tom. Controlling Software Projects: Management, Measurement and Estimates. Upper Saddle River, NJ: Prentice Hall PTR, 1986.
5. Banerjee, Guntam. "Use Case Points – An Estimation Approach." Aug. 2001 <www.comp.nus.edu.sg/~bimlesh/oometrics/15/1035194512861.pdf>.
6. Cockburn, Alistair. "Use Cases, Ten Years Later." Software Testing and Quality Engineering Magazine Mar./Apr. 2002.

About the Author

Arlene F. Minkiewicz is the chief scientist at PRICE Systems, LLC. In this role, she leads the cost research activity for the entire suite of cost estimating products that PRICE provides. Minkiewicz has more than 24 years of experience with PRICE building cost models. Her recent accomplishments include the development of new cost estimating models for IT projects. Minkiewicz has published many articles on software measurement and estimation and frequently presents her research at industry forums.

PRICE Systems, LLC
17000 Commerce PKWY
STE A
Mt. Laurel, NJ 08054
Phone: (856) 608-7222
E-mail: arlene.minkiewicz@pricesystems.com

Understanding Software Project Estimates

Katherine Baxter
Champlain College

With two-thirds of software projects running long and over budget [1], *it is important that upper management understand the value of proper estimation techniques, and that their estimators are as accurate as possible. This article discusses formal estimation techniques, accurate software estimation tools, the misinterpretation of estimation as target setting, and the accuracy of estimates.*

Software projects are notoriously complex and difficult to estimate accurately. Many authors have referred to estimation as a *black art*. Estimating a project accurately involves carefully analyzing data from many different aspects of the project, and using a number of different techniques to get the best estimate possible with the given information. Even then—depending on the accuracy of project information and how far along the project is in its life cycle—the estimate may still not be very close to the actual values at the project's completion.

Accurate project estimates, even early in a project's life cycle, are extremely important to an organization's success. For example, when prospective customers have received their requested proposals for a certain project and have gone through their initial proposal evaluation process, those *short-listed* organizations will have to submit cost and schedule estimates. If the organization awarded the project has initial estimates that are too optimistic, they may get stuck developing a project that runs over budget or breaks the contract in terms of scheduling; the end result may be losing revenue instead of turning a profit. If a certain organization's initial estimate is too pessimistic, they are likely to be rejected in favor of another whose estimates look more favorable to the customer. When so much is resting on finding accurate estimates—especially early in the requirements definition phase of a project—it is in every company's best interest to apply any and all available techniques to make sure their estimates are as close as possible to the actual values.

Despite many developments in estimation techniques, most project estimates are still not very good. In fact, only about one-third of all projects are completed on-time and on-budget, with some off in both areas and others so far gone that they are discontinued before completion. These numbers represent huge losses in profit. However, it is sometimes difficult to trace these losses directly back to problems in the estimating processes.

A number of different factors play into this problem of poor estimation. One of the most common is that, in many cases, accurate estimation techniques are simply not applied. Many organizations, in the interest of saving time and money, try guessing at project estimates with no formal process for determining the project's cost and scheduling needs [2]. This is most likely due to a lack of understanding of the importance of estimates. Estimating accurately involves a lot of time and money, and there is not always any direct and easy way to see a return on any investments made in proper estimation processes.

Sometimes organizations will try to take shortcuts to save money in estimation. One of the most common mistakes is basing estimates entirely off of historical data from past projects. The estimator will simply find a project that seems similar to the project they are estimating and use the final values from that project. While this technique is an important piece of the estimation process, when used alone it is a proven cause of schedule and cost overruns [2]. Since the industry is constantly developing and changing quickly, estimates based entirely on historical data are not enough. For accuracy, historical data should be used alongside other methods to find the most accurate possible estimates. It is important to use those values, but also to check them against values obtained from the latest updated estimation models and tools to keep up with the industry as it changes.

Another common problem is that deadlines are sometimes set before estimates are even made. The estimator then has to count backwards from the deadline in order to make the estimate instead of forming their own schedule [1]. This discourages a proper estimating process and is likely to influence the estimator into making overly optimistic estimates.

Even when good estimating models and tools are used, sometimes not enough effort is put into making sure the estimates are based on accurate or complete information. As one author puts it, "any estimate is only as accurate as its least accurate input variable" [1]. It is extremely important that estimators take the time to make sure all their data is as accurate as possible for that stage of the project's development before committing to any final estimates. If the data used to make the estimates is incomplete or inaccurate, the resulting estimate will also be incomplete and inaccurate.

There are a number of tactics that can be applied to help solve some of these problems. The most important of these is that a proper formal estimation process should be used in all cases. Guessing or taking shortcuts to come up with quick results will not provide the quality estimates that are needed. Formal estimation techniques are proven to make estimates much more accurate, especially early in project development when the requirements and risks are not yet clearly defined [2]. When so much rests on estimates being accurate, it is important not to cut any corners.

Another technique is to use multiple estimating techniques in combination. Using historical data from your own organization (as well as from others) and a few different models and tools can help find the best possible estimate [3]. Each of the estimates found from these techniques is an approximation. It is very unlikely that any of them will be exact: Multiple techniques can bring the final estimate closer to the right target.

Since all estimates are really only approximations, it is best to avoid settling on any single number. Early on—when requirements may not be completely defined and other changes in the project's completion plans may still take place—estimates are often significantly inaccurate. One technique to improve accuracy is to provide a best-case-scenario estimate, a worst-case-scenario estimate, and an expected estimate. The final estimate can then be provided as a range of values, rather than committing to one specific value from the start [1]. For example, an estimator may say that a project is likely to cost between $350,000 and $600,000. This gives the organization a general idea of how much the project will cost, but does not fool them into thinking the estimate is more than an approximation.

For large-scale projects, however, manual estimation is not enough. There are a number of complex tools available to help create accurate estimates; for complicated projects with many parts, these are sometimes the only way to get a good estimate. In general, these tools are much more accurate than estimation by hand. Manual estimates tend to be too optimistic, often by more than 30 percent [4]. Even when they are done more carefully, a manual estimate will rarely be more accurate than an estimate made with the help of one of these tools.

Understanding the importance of accurate estimation—and a willingness to put in the resources needed to support good estimation processes—are vitally important to an organization's success. Learning what is involved in making good estimates, and what techniques can help improve estimate accuracy even more, are important first steps.

Estimation Tools

There are a number of software estimation tools available to help in making accurate estimates. For larger projects of more than a thousand function points, it is almost always necessary to use these tools rather than attempt manual estimation: They are complex and have so many factors that must be taken into account that an estimation tool is much faster than estimating by hand [4]. Even for smaller projects, there are a number of advantages to using automated tools rather than pure manual estimation.

Automated estimation tools tend to be much more accurate than manual estimation. Since there is less human interference, the estimates are less likely to be influenced by human bias that might make them unrealistically optimistic. Automated tools are also much less likely to underestimate or overlook effort involved in areas such as design, documentation, and testing [5].

There are many features that are fairly standard across most estimation tools that will assist in sizing estimates and estimating at the phase, activity, and task levels. They will also help with general quality and reliability estimation. Most will support size measurements in both function point and source lines of code units, and support conversions from one to the other. Most tools also have support for a variety of different languages, including older languages such as COBOL and FORTRAN.

Some tools also provide other features, but these are not standard across all tools. Some will perform risk and value analysis, inflation calculations for long-term projects, and currency conversions for international projects. Some will provide support for various standards like the SEI's CMMI®. Many tools also allow the organization to input historical data, which is then used to adjust scheduling and other estimates. Because estimation tools are often used in combination with project management tools such as Artemis or Microsoft Project, many also provide interfacing capabilities [4].

Even with all these features, there are still many aspects of a project that must be accounted for manually. They include things such as fees for trademark and copyright searches and legal expenses for any breach of contract if a project is not completed on-time or on-budget [4]. Generally though, when all input factors are carefully considered and are accurate, estimation tools will provide a thorough and accurate estimate of a much higher quality than can be achieved by hand.

"Understanding the importance of accurate estimation, and a willingness to put in the resources ... are vitally important to a company's success."

Estimation Isn't Target Setting

One common misunderstanding is that estimation is simply the process of finding a target end-date and total cost for a project. This assumption is dangerous because estimates are not exact, and early estimates are not nearly precise enough to pin down exact values. Committing to specific values early on is likely to cause the project to either go over schedule or run too long.

Estimates can help provide a general idea of when a project will be finished and how much it will cost, but it is impossible to settle on specific values with total certainty. For example, one could estimate that a project is likely to be finished sometime between 10 and 16 months, but there is still a chance that the project will not be completed within this time frame. Even setting a target end-date at the end of this range could be dangerous. Moving the target end-date with no particular purpose does nothing but change the probability of the project actually finishing on schedule [2]. Clearly, this estimate is not a good basis for choosing a specific end-date.

Target setting takes place when an external factor is determining a required end-date or budget. If a project must be completed by a certain event (like a specific conference) or by the end of a fiscal year, then a target end-date is set. The estimation process then becomes a matter of deciding how much can be completed by that date with a certain amount of resources, and refining the requirements to fit into this time frame [2]. Working backwards carefully like this can ensure that the project finishes by a certain date, but it may involve sacrifices in requirements or an increase in resources. If there is no specific end-date, using the estimation process to set one is risky.

Estimation is a process of taking a set of input values, conducting some careful analysis, and ending with a set of results. Once these results are found, it makes no sense to try to argue against or change results. The estimation process cannot be altered to achieve different results, and trying to change them to fit within a certain budget or time frame will only make the project less likely to succeed. If the results of an estimate are not satisfactory, the only reasonable way to change them is by adjusting the inputs—the information that the estimate was based on.

Inputs, like requirements and resources, determine the final results of estimation. Changing these values is the only way to reasonably affect the estimation results. If the estimated cost of a project is too high, it may be possible to adjust it by reducing the functionality of the system, consequently making the project smaller. If the estimated time frame is too long, it can often be changed by adding more resources to the project. These are the only meaningful ways to change estimated values.

Accuracy of Estimates

Early on in a project's development, estimates are likely to be very inaccurate. Estimates are based on inputs concerning what is known about the project; consequently, if the concept of the project is not entirely clear or does not match exactly with what the finished product will look like, the estimate is likely to be pretty far off from the final values. Functionality may need to be added or adjusted as the product is developed because requirements change often during project development and systems often don't work exactly as planned. Because this uncertainty is based in the inputs and not in the estimation process, spending more time on the estimate itself will not necessarily make it more accurate [6]. While spending extra time in the planning phase and ensuring all of the processes used are mature can help increase the accuracy of early estimates, it is still not accurate enough to meaningfully commit to specific values [2].

As the project progresses—and requirements become clearer and eventually *set in stone*—estimates become more accurate. Estimates in the conceptual phase (before extensive planning is done) are often off by as much as 50 percent, but are down to approximately 25 percent by the time functionality is determined. By the time the project is actually being implemented, estimates are usually within 10 percent of the final values [7]. It is important to note that these values are based on estimates made by skilled, experienced estimators and formal estimation methods. If the estimators were less experienced or used less precise estimating techniques, their estimates would likely be even further off the mark [6].

The most significant improvements in estimation accuracy occur during the first 20 to 30 percent of project development [6]. This represents the planning phases where the unknowns that cause such problems with estimation accuracy are being eliminated. To understand an estimate at any given point in development, it is important to understand how precise one can expect estimates to be at that stage.

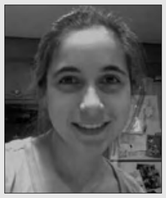
Because early estimates are so inaccurate, it is important that they are never treated as exact expected final values. Early estimates should always be expressed in ways that clearly show this uncertainty, such as describing them as ranges of values rather than fixed points [2]. It is also essential that no commitments to specific values are made early on. Because there is so much possibility for inaccuracy early in development, any commitments made within the first 30 percent of a project's life cycle are not reasonable or meaningful [6].

Estimates should never be treated as exact final values, but early estimates in particular should be treated with care. As one author explains, "the only time we have sufficient data to truly warrant the label 'accurate' is at the very end of the project when all the variables are resolved. Unfortunately, no one will ever ask for an estimate at that stage" [8].◆

References

1. Dekkers, Carol A. "Creating Requirements-Based Estimates Before Requirements Are Complete." CrossTalk Apr. 2005 <www.stsc.hill.af.mil/crosstalk/2005/04/0504Dekkers.html>.
2. Henry, David. "Software Estimation: Perfect Practice Makes Perfect." CrossTalk June 2002 <www.stsc.hill.af.mil/crosstalk/2002/06/henry.html>.
3. Stutzke, Richard. "Software Estimation: Challenges and Research." CrossTalk Apr. 2000 <www.stsc.hill.af.mil/crosstalk/2000/04/stutzke.html>.
4. Jones, Capers. "Software Cost Estimation in 2002." CrossTalk June 2002 <www.stsc.hill.af.mil/crosstalk/2002/06/jones.html>.
5. Jones, Capers. "Software Cost Estimating Methods for Large Projects." CrossTalk Apr. 2005 <www.stsc.hill.af.mil/crosstalk/2005/04/0504Jones.html>.
6. "The Cone of Uncertainty." Construx. 2008 <www.construx.com/Page.aspx?hid=1648>.
7. Roetzheim, William. "Estimating and Managing Project Scope for New Development." CrossTalk Apr. 2005 <www.stsc.hill.af.mil/crosstalk/2005/04/0504Roetzheim.html>.
8. Armour, Phillip. "Ten Unmyths of Project Estimation." Communications of the ACM Vol. 45. No. 11, Nov. 2002.

About the Author



Katherine Baxter is a programmer at Champlain College's Emergent Media Center, where she creates computer games for use in education and training. Baxter also recently developed educational games for children at Tertl Studos in Montpelier, Vermont. She is completing her bachelor's degree in software engineering at Champlain College, where she will graduate this spring.

Champlain College
PO Box 670 Box 51
Burlington, VT, 05401
Phone: (978) 807-1793
E-mail: baxter.katherine@gmail.com

STC
Systems & Software
Technology Conference

"Technology: Advancing Precision"
20-23 April 2009 - Salt Lake City, Utah

120+ TECHNICAL PRESENTATIONS
Beginning to Highly Advanced

TRAINING AND CERTIFICATION OPPORTUNITIES AT A REDUCED COST:
International Software Process Improvement Certification (ISPIC™)
Prepare for SE Certification with an INCOSE Tutorial
IEEE Certified Software Development Professional (CSDP)

COLLABORATIVE NETWORKING
Between Military/Government, Industry, and Academia

TRADE SHOW
Featuring Cutting-Edge Technologies Beneficial to Your Organization

SCENIC LOCATION
Salt Lake City, Utah, a major metropolitan city with the warm, welcoming friendliness of a small western town. Mild Spring weather allows for skiing in the mountains and golf in the valley.

REGISTER TODAY!
WWW.SSTC-ONLINE.ORG

DAWIA Continuous Learning Point (CLP) Credits:
DoD acquisition personnel who are SSTC speakers or attendees may be able to use their SSTC experience to satisfy part of their acquisition requirement to accrue 40 Continuous Learning Points (CLPs) annually. Please consult your organization's training monitor.

www.sstc-online.org

Real Science ... Fiction

It was a quiet afternoon. I was in my home-office busily penning the masterpiece that was to become my first novel when a bizarre, pulsating noise began emanating from the computer speakers. It buzzed in a strange symphony for a few moments, faded, and was gone.

My mind immediately became engaged in rampant speculation. The fact that I lived near a large military base weighed in and possibilities swirled as to what covert operation or top secret data burst I had unintentionally intercepted. But then common sense and years of communications engineering experience weighed in.

When that raspy buzz returned about a week later, I opened the blinds to see a natural gas company truck, bristling with antennas, slowly driving by. The sound was nothing more than circuitry in my computer speakers partially demodulating an active data acquisition transmission used to gather usage information from gas meters equipped with digital radio frequency technology.

Science fiction writer Arthur C. Clarke said: "Any sufficiently advanced technology is indistinguishable from magic." The statement implies that when our ability or desire to comprehend science goes beyond what we believe to be possible, we limit ourselves by our beliefs, and not necessarily technology. If you think about it, it's often only a lack of knowledge or understanding that makes one man's science another's science fiction.

Did you know that during World War II, the Japanese military made "balloon bombs" designed to travel the jet streams across the Pacific to drop on the United States? A few actually made it. The Japanese were also working on a giant high radio frequency "ray gun" intended to immobilize attacking troops en masse. As well, the Germans were working on a host of their own strange weapons, including a giant air cannon intended to blow the wings off of overflying aircraft.

A wild American idea that lost out to the Manhattan Project was the "Bat Bomb." Thousands of bats were fitted with small incendiary devices and loaded onto trays in a layered "bomb" to be dropped over Japanese cities. Theoretically, the trays would deploy and the bats would disperse, landing and roosting in the mostly wooden Japanese structures where the incendiaries would ignite and burn, in-turn setting the cities ablaze. The weapon was built and tested, but never deployed.

During the Cold War, the Soviet Union was a candy store for supposed "fringe science." Reports of psychic warfare involving experiments with mind control, telekinetics, and remote-viewing espionage (the ability to psychically "see" the physical environment of a distant location) were public knowledge, or hyperbole. In the '70s, a Nikola Tesla-like glowing "energy dome" was supposedly seen over Siberia by a commercial airline pilot. A reconnaissance photo published in 1980 showed a facility speculated to be their government's attempt at a massive particle beam weapon. Soviet agents were once caught beaming microwave signals into the American Embassy. Some say it was surveillance. Others say it was an ongoing mind-control experiment. What is real?

I wasn't quite a teenager in the mid-70s when the infamous Russian Duga series over-the-horizon radar system suddenly filled the world's airwaves and the speakers of my modest Electrophonic stereo with its sharp, pulsing, 10 cycles per second "TAP TAP TAP." Designed to detect the plumes of long-range ballistic missile launches over the lands of its enemies, the signal wreaked havoc with communications and electronic systems throughout North America where a primary signal path had been directed from over the North Pole. Conspiracy theorists went berserk. But it was the world's Amateur Radio operators who quickly pulled out their oscilloscopes and radio direction finding equipment to evaluate the signal, find its origin, and posit as to its likely and logical intent.

Chris Carter's TV series, *The X-Files*, coined a phrase equally prophetic to conspiracy theorists, scientists, and Tibetan monks: "The truth is out there."

There are those, possibly a few reading CrossTalk right now, who are privy to real science and technology so advanced that for many it would undoubtedly seem like science fiction—or magic. Even the ideas of particle beam weapons and "death rays" have now evolved into ground- and aircraft-based chemical lasers and microwave-based crowd deterrents. Technology, creativity, and ambition soldier on.

So what else is "out there"?

Now that I think about it, I can't be sure that really *was* a company gas truck. Did some covert agency find out about those strange scalar weaponry documents that mysteriously showed up a few years ago?

What could really be going on inside all those new digital televisions and converter boxes that we "have to get"? Can the microphone and camera in my cell phone really be activated without my knowledge? Is the European Organization for Nuclear Research (CERN) really producing antimatter? Could the mass energy in a fountain pen actually supply enough power to run a city for a day? The Defense Advanced Research Projects Agency? The Aurora Project? Dark matter? The High Frequency Active Auroral Research Program? Wormholes?

And just how do you suppose Nikola Tesla's scientific papers managed to become "lost" anyway?

Isn't paranoia—I mean science and imagination—wonderful?

—Ranse Parker
author@ranseparker.com

Can You BackTalk?

Here is your chance to make your point without your boss censoring your writing. In addition to accepting articles that relate to software engineering for publication in CrossTalk, we also accept articles for the BackTalk column. These articles should provide a concise, clever, humorous, and insightful perspective on the software engineering profession or industry or a portion of it. Your BackTalk article should be entertaining and clever or original in concept, design, or delivery, and should not exceed 750 words.

For more information on how to submit your BackTalk article, go to <www.stsc.hill.af.mil>.

CROSSTALK / 517 SMXS/MXDEA
6022 Fir AVE
BLDG 1238
Hill AFB, UT 84056-5820

PRSRT STD
U.S. POSTAGE PAID
Albuquerque, NM
Permit 737

Homeland Security

Software Assurance Program

Software is essential to enabling the nation's critical infrastructure. To ensure the integrity of that infrastructure, the software that controls and operates it must be reliable and secure.

Visit https://www.us-cert.gov/SwA to learn more about the software assurance program and how you can become more involved.

Security must be "built in" and supported throughout the lifecycle.

Visit https://BuildSecurityIn.us-cert.gov to learn more about the practices for developing and delivering software to provide the requisite assurance.

Sign up to become a free subscriber and receive notices of updates.

BuildSecurityIn

Software Assurance
Community Resources and Information Clearinghouse

Software Assurance Working Groups

The Department of Homeland Security National Cyber Security Division has established stakeholder working groups to support each of its Software Assurance Program's focus areas. The goal of the working groups is to continue to support Software Assurance initiatives by bringing together members of government, industry, and academia with vested interests in software security to discuss and promulgate applicable practices and methodologies.

Software Assurance Focus Areas and Working Group Matrix

https://buildsecurityin.us-cert.gov/swa/

The Department of Homeland Security provides the public-private framework for shifting the paradigm from "patch management" to "software assurance."

CROSSTALK thanks the above organizations for providing their support.